Contribution to the Analysis of Discrete Event Systems

Hervé Marchand

HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et Communication

par

Hervé Marchand

Contribution to the Analysis of Discrete Event Systems

soutenue le 6 juin 2017 devant le jury composé de

Bátrice Bérard	Rapportrice
Alessandro Giua	Rapporteur
Thierry Jéron	
Jean-Jacques Lesage	
Sophie Pinchinat	
Jean-Francois Raskin	Rapporteur
Olivier H. Roux	

Contents

Introduction

Since the 90's, computer systems take a growing place into our everyday lives. It might be embedded-systems, such as in robotic, automotive or avionic systems, telecommunication or transportation systems or energy services, etc. The presence of such systems offers new possibilities, but the price to pay is the increasing software failures which can have dramatic consequences in terms of human lives or prohibitive costs. Manual validation is expensive, may be impossible for large systems, and is permeable to mistakes. The development of automatic tools serving to analyze or to ensure security/safety has thus become crucial to discover and avoid breaches and mistakes in the development of embedded systems.

One can generally decompose validation methods into three parts: modeling, specification, and validation: the first step consists in describing the system that must be verified. Generally, this description is either given by the source code of the system, or by a formal (i.e., mathematical) model extracted from the system. This model can be an hybrid system [ACH+95], an automaton [ASU86] and their extensions (stochastic, timed, etc), a Kripke structure [Kri63], etc. The specification step consists in describing the properties or requirements that the system must satisfy. These properties can be specified by predicates, by formulas of a temporal logic such that the linear time logic (LTL) [Pnu77] or the branching time logic (CTL) [CE82]. The third step is the validation, where the model is checked against the specification. Various techniques exist to validate the correctness of a system or to detect errors in it. Hereby, we focus on model-based techniques that we list below.

Model-Checking A classical and well-known technique to validate a model of a system is given by model checking. It has been introduced in [CE82] and [QS82]. This method works on a model (specification) $\mathcal{G}$ of the system which is generally given by a (finite) state transition system like Kripke Structure.

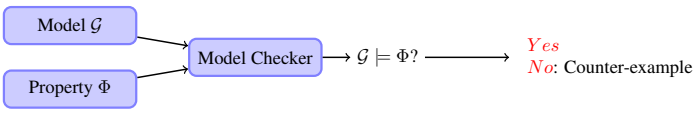


Figure 1: Model-checking principle

The specification Φ, which gives the properties that the system must satisfy, is generally specified by a formula of a temporal logic such that the linear time logic (LTL) or the branching time logic (CTL). Model checking techniques analyze the model of the system

and produce an output which says if the model satisfies the given specification (see Fig. 1). When the specification is not satisfied, a witness (e.g., a sequence of events leading to the violation of the specification), which shows that the model of the system does not behave correctly, is given. This proof is often given as an error trace and a human assistance is then required to identify the source of the error and to fix it on the specification.

On-line Monitoring. Even-though validated by model-checking, the actual system usually derived from the specification and can be subject to errors w.r.t. the implementation. So, there is still a need to validate on-line the behaviour of the system.

- **Model-Based Testing.** Testing aim is to validate at run-time an implementation $\mathcal{I}$ during its execution according to some requirements [Bri88, Tre96a, CJRZ02],[J15]. It is an alternative to the model-checking whenever the model checking algorithms are prohibitive in terms of complexity or whenever the expected requirements has to be tested on the actual implementation rather than on its model.

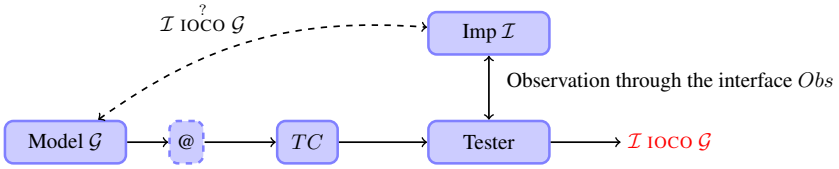


Figure 2: Model-based conformance testing

Among the different testing theories, one can consider the model-based conformance testing [Tre96b] which consists in comparing the observable behavior of a black-box implementation $\mathcal{I}$ of a system with respect to those allowed by its formal specification $\mathcal{G}$ representing the expected behavior of $\mathcal{I}$ (expressed in a "mathematical" framework). The idea is to derived from the specification $\mathcal{G}$ a set of test cases TC that are run on the implementation in order to detect some deviations between the implementation and its model. If a difference (a non-conformance) is detected, the implementation has to be modified (typically, bugs have to be fixed) and the process is iterated until no more errors are detected.

A classical formalization of conformance is called **ioco** [Tre96b] which defines what it means for an implementation of a system to be correct w.r.t. a specification. Intuitively, an implementation conforms to its specification if after each observable trace[1] of the specification, the implementation only exhibits outputs and blockings that are also allowed by the specification.

- **Run-time Verification.** When the model is not available or too complex to be formally checked, it is still possible to validate on-line the expected specifications. At an abstract level, a run-time verification approach consists in synthesising a verification monitor (cf. [HR02, FFM12]), i.e., a decision procedure used at run-time on the

[1] Not all but only a part of the actions of the implementation can be observed.

implementation $\mathcal{I}$. The monitor (partially) observes the system under scrutiny and emits verdicts regarding the satisfaction or violation of the property of interest.

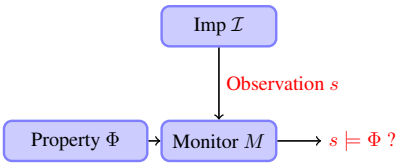


Figure 3: Run-time Verification principle

Diagnosis. Besides model-checking and conformance model testing that are traditionally used to verify the properties that hold on the model of the system, one can be interested in identifying on-line that some particular behaviours (called faults) occur on the actual system $\mathcal{I}$ that is described by a formal model $\mathcal{G}$ of the system (note that $\mathcal{G}$ is supposed to be correct w.r.t. $\mathcal{I}$ ($\mathcal{G} \sim \mathcal{I}$ in Fig 4[2]), and that $\mathcal{G}$ includes the fault models). Diagnosing is an increasingly active research domain and model-based approaches have been proposed which differ according to the kind of models or faults they consider [SSL+96, BLPZ98, RC98, PC05, DLT00a, FBJ+00]. Most of the times, diagnosis aims at achieving the detection of faults that might have occurred in the system. Detection is an oracle (called diagnoser) that decides whether the system works in normal conditions or whether a fault happened (knowing that only a part of actions that occur in the system can be observed). The diagnoser is derived from the model of the system and the fault that one wants to detect or isolate (a fault might be the occurrence of a set of actions, a sequences of actions, or in general given by an extension closed languages). A system is said to be diagnosable if the diagnoser can accurately detect faults a finite number of steps after their occurrence.

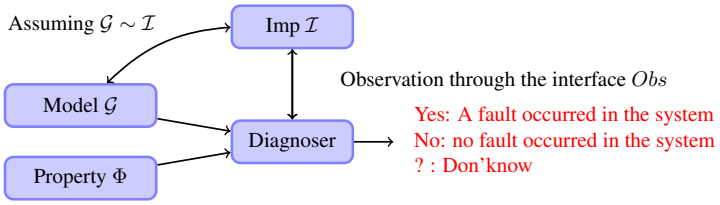


Figure 4: Diagnosis technique

Supervisory Control. Compared to model-checking or monitoring, the supervisory control problem is to correct the model of the system w.r.t. some requirements. The control theory of discrete event systems [RW89, CL08] allows the use of constructive methods

[2]This constitutes a main difference with the conformance testing where the behaviour of the model is compared with the implementation in order to find some differences or deviations.

ensuring, a priori and by means of control, required properties on a system's behavior. Supervisory control can be used to fix errors specifications, or in the design process to obtain correct-by-construction software's (like for a refinement process). The starting point of these theories is: given a model of the system $\mathcal{G}$ (that is supposed to correctly represent the behaviour of an implementation) and control objectives Φ (modelled as formal properties expressed in LTL or CTL w.r.t. the specification), a controller $\mathcal{C}$ must be derived by various means (based on model checking techniques) such that the resulting behavior of the closed-loop system meets the control objectives. This controller can be applied in parallel with specification (in that case, the controller can be seen as a refinement of the specification), or coupled with the implementation $(\mathcal{I})$ in a way to correct an already deployed system whenever the requirements is evolving.

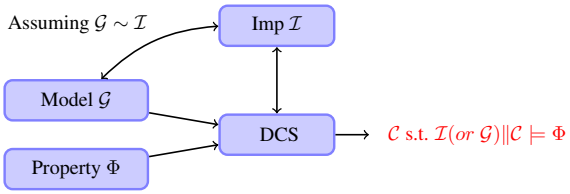


Figure 5: Discrete Controller Synthesis (DCS) Principle

Enforcement. Runtime enforcement [Sch00a, HMF06, LBW09, FMFR11] is a verification/validation technique aiming at correcting (possibly incorrect) executions of a system of interest, which is supposed to be unknown (as for the run-time verification). In traditional approaches, the enforcement mechanism is a monitor E that implements a decision procedure that take inputs from the implementation, possibly corrects them and outputs a sequence of events satisfying the property. . How a monitor transforms the input sequence is done according to a high-level specification, formalized as a property, that indicates correct and incorrect sequences. Moreover, a monitor should only output correct sequences (the monitor is sound) and should minimally alter the input sequence (the monitor is transparent).

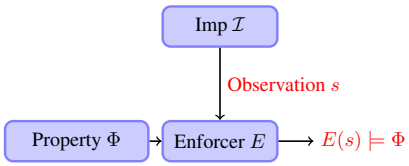


Figure 6: Enforcer Mechanism

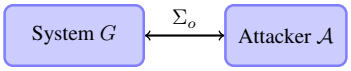
Contributions: we now summarize our contributions in the analysis of (embedded) systems modeled by (finite) transition systems.

- In the field of diagnosis of discrete event systems, we propose in Chapter 2 a model of supervision patterns general enough to capture past occurrences of particular trajectories of the system. Modeling the diagnosis objective by a supervision pattern allows us to generalize the properties to be diagnosed and to render them independent of the description of the system. In [C22], we proposed techniques for the verification of the diagnosability and for the construction of a diagnoser based on standard operations on transition systems. We have shown that these techniques are general enough to express and solve in a unified way a broad class of diagnosis problems found in the literature. We recently extended these results to a class of infinite recursive system (*Recursive Tile Systems*) [J4],[C11]. Further, this work has been extended for the prediction of supervision patterns. The occurrences of the pattern are predictable if it is possible to infer about any occurrence of the pattern before the pattern is completely executed by the system [C19]. Recently, we examined the case of transient faults, that can appear and be repaired. Diagnosability in this setting means that the occurrence of a fault should always be detected in bounded time, but also before the fault is repaired. Checking this notion of diagnosability is proved to be PSPACE-complete [C1].

- In Chapter 3, we address the supervisory control problem based on the Ramadge& Wonham framework [RW89]. Our contribution in this field was to tackle the problem of the control of concurrent systems composed of multiple sub-systems. The idea was to compute controllers locally without having to compute the global system. We have adopted a language-based approach and proposed novel notions and algorithms in order to solve this problem [J14]. *This works has been done in the context of Benoit Gaudin's PhD thesis* [Gau04].

  Following this work, we have also been interested in the control of distributed systems communicating asynchronously; the aim was to build local controllers that restrict the behavior of a distributed system in order to satisfy a global invariance property [J6], [C12]. Distributed systems are modeled as *communicating finite state machines* with reliable unbounded FIFO queues between subsystems. To refine their control policy, controllers can use the FIFO queues to communicate by piggybacking extra information to the messages sent by the subsystems [C13]. We define synthesis algorithms allowing to compute the local controllers and ensure termination by using abstract interpretation techniques, to over-approximate queue contents by *regular languages* [J6]. *This work was part of the PhD of G. Kalyon, with Th. Massart (ULB, Bruxelles) and T. Le Gall (CEA)* [Kal10].

- In the field of computer security, a problem that received little attention so far is the enforcement/detection of confidentiality properties. As a confidentiality property, we consider the notion of *opacity* (introduced by [BKMR08] which is a very general notion modeling the absence of information flow towards inquisitive attackers. This information flow might occur when a configuration of the system is reached, an event occurred or more generally when the system executes a particular sequence of actions. In Chapter 4, we consider the problem of information flow in the case where the system is given by a finite transition system $\mathcal{G}$ and an inquisitive user $\mathcal{A}$, called the adversary which partially observes the system through a particular interface.



We provides various techniques allowing to

- model-check the opacity property [J9];
- detect opacity violations at run-time using diagnosis techniques [C18];
- enforce the opacity property on a critical system. In particular, we study how one can restrict behavior of a system in order to avoid information leakage using supervisory control theory [J12];
- enforce at run-time the opacity property [J9].

This work has been realized in the context of Jérémy Dubreil's PhD Thesis, under the co-supervision of T. Jéron [Dub09].

- We finally present recent works related to the on the fly enforcement of real-time properties, *in the context of Srinivas Pinisetty's PhD Thesis, under the joint supervision of T. Jéron and Y. Falcone* [Pin15]. Runtime enforcement is a verification/validation technique aiming at correcting possibly incorrect executions of a system of interest. In this chapter, we consider enforcement monitoring for systems where the physical time elapsing between actions matters. Executions are thus modelled as timed words (i.e., sequences of actions with dates). We consider run-time enforcement for timed specifications modelled as timed automata. Our enforcement mechanisms have the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus possibly allowing for longer executions. To ease their design and their correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that sfpecify the input-output behaviour in terms of transformations of timed words, constraints that should be satisfied by such functions, enforcement monitors that describe the operational behaviour of enforcement functions, and enforcement algorithms that describe the implementation of enforcement monitors. The feasibility of enforcement monitoring for timed properties is validated by prototyping the synthesis of enforcement monitors from timed automata [J1],[J2],[J5], [C3], [C4], [C8]. This part is given by the article [J1] provided with this document.

Other Contributions: This introduction ends with a list of contributions not presented in this document:

- **Control of symbolic systems.** We consider here reactive synchronous systems, i.e., data-flow systems reacting to inputs sent by the environment, and producing outputs resulting from internal transformations. In this framework, part of the inputs is uncontrollable Y_{uc} (it may correspond to measures from sensors), whereas the other part of inputs Y_c (e.g., user commands to actuators) is controllable. Triggering these events makes the system evolve from a configuration to some other one. In this setting, we aim at synthesizing controllers restricting the possible values of the controllable inputs so as to ensure some properties.

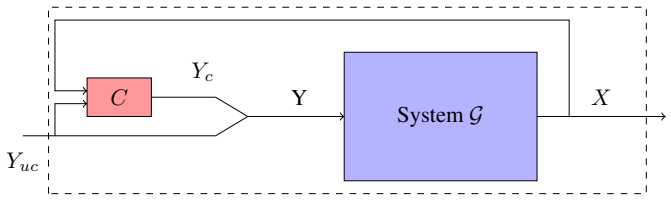


Figure 7: Controlled Symbolic transition system.

In this framework, discrete controller synthesis is an operation that applies on a transition system (originally uncontrolled) with a given control objective: a property that has to be enforced by control. In this work, we consider invariance and reachability of a subset of the state space (typically, forcing a predicate over the state variables of the system to be always true or to be true in the future). But we can also use observers composed in parallel with the original system, to enable general safety properties[3].

The purpose of control synthesis is then to obtain a controller, which is a constraint on values of controllable variables Y_c, as a function of the current state and the values of uncontrollable inputs Y_{uc}, such that all remaining behaviors satisfy the objective. As the synthesized controller is maximally permissive, it is *a priori* given by a relation which can be transformed into a control function by adding new constraints between variables. This is illustrated in Figure 7, where the transition system, as yet uncontrolled, is composed with the synthesized controller C, which is fed with uncontrollable inputs Y_{uc} and the current state value from S, in order to produce the values of controllable Y_c which are enforcing the control objective. The transition system then takes $Y = Y_{uc} \cup Y_c$ as input and makes a step by computing the new state and producing the new outputs.

- When all the variables of the systems are Boolean, we have shown how to compute controllers so that a logical property is verified on the controlled system. We also developed symbolic algorithms allowing to solve optimal control problems [J18], [C30]. All these techniques have been integrated in the Polychrony environment [J18] and the Heptagon Language[J8] and more particularly in the Sigali tool box [J18]. This tool has been successfully used to verify or control various academic or industrial systems (the incremental design of a power transformer station controller [J18], [J17], [J19]) as well as non trivial examples from robotic control-command systems [J13]. A similar approach was applied for the control of reconfigurations in fault-tolerant systems [C15],[C21].
- During the last past years, we extended these results to systems handling numerical variables. The principle is the same except that the state space of the system is now infinite. We provide algorithms allowing to compute a controller ensuring safety properties. Since variables in infinite domains are ma-

[3] An observer is simply an LTS allowing to capture a safety property over the sequences of the system (e.g. the event a does not occur twice in a row in the system). As usual, we assume that an observer is complete so that its composition with the system let the system behavior unchanged.

nipulated, these algorithms are based on abstract interpretation techniques that over-approximate the state space that has to be forbidden by control, hence perform the computation on an abstract (infinite) domain (typically polyedra) [C7], [C5], [C2]. This methodology has been implemented in the tool Reax whose aim is to replace Sigali, which is limited to finite state systems described by boolean variables.

This topic corresponds to a *long-term collaboration with E. Rutten as well as with two post-doc researchers: N. Berthier and G. Delaval.*

- **Test of reactive systems.** In the field of automatic test generation, we have been interested in combining formal verification and conformance testing. A specification of a system, an extended input-output automaton, which may be infinite-state-and a set of safety properties (”nothing bad ever happens”) and possibility properties (”something good may happen”) are assumed. The properties are first tentatively verified on the specification using automatic techniques based on approximated state-space exploration, which are sound, but, as a price to pay for automation, are not complete for the given class of properties. Because of this incompleteness and of state-space explosion, the verification may not succeed in proving or disproving the properties. However, even if verification did not succeed, the testing phase can proceed and provide useful information about the implementation. Test cases are automatically and symbolically generated from the specification and the properties and are executed on a black-box implementation of the system. The test execution may detect violations of conformance between implementation and specification; in addition, it may detect violation/satisfaction of the properties by the implementation and by the specification. In this sense, testing completes verification [J15], [C26].

Chapter 1

Notations

In this chapter, we provide notations and basic results that will be used throughout this document. We first introduce the notion of languages and further model of the labeled transition system (LTS) as well as some basic transformations associated to this model.

1.1 Languages

We start first by recalling useful standard notations: we assume given an alphabet Σ, that is a finite set $\{\sigma, \sigma_1, \ldots\}$. The set of finite words over Σ is denoted by Σ^*, with ϵ for the empty word. In the paper, typical elements of Σ^* are $s, t, u, \ldots$. For each $s, t \in \Sigma^*$ of the form $s = \sigma_1 \ldots \sigma_n$ and $t = \sigma'_1 \ldots \sigma'_m$ $(n, m \in \mathbb{N})$, the *concatenation of s and t* is still a word defined by $s.t = \sigma_1 \ldots . \sigma_n . \sigma'_1 \ldots \sigma'_m$. The *length* of $s \in \Sigma^*$ is denoted $|s|$ (the length of the empty word is zero). We let Σ^n with $n \in \mathbb{N}$ denote the words of length n over Σ. A set $\mathcal{L}$ of finite words over Σ, $\mathcal{L} \subseteq \Sigma^*$, is called a *language* over Σ.

Given two words $s, s' \in \Sigma^*$, we say that s' is a *prefix* of s whenever there exists $s'' \in \Sigma^*$ such that $s = s'.s''$. The set of prefixes of a language $\mathcal{L} \subseteq \Sigma^*$ is given by $pref(\mathcal{L}) = \{s \in \Sigma^* : \exists s' \in \Sigma^*, s.s' \in \mathcal{L}\}$. $\mathcal{L}$ is said *prefix-closed* if $pref(\mathcal{L}) = \mathcal{L}$ and *extension-closed* if $\mathcal{L} = \mathcal{L}.\Sigma^*$. Given a word $s \in \Sigma^*$, the (right) *residuation* of s w.r.t. $\mathcal{L}$ and $\Sigma' \subseteq \Sigma$, noted $s^{-1}(\mathcal{L}, \Sigma')$ consists of all words s' over Σ' such that $s.s' \in \mathcal{L}$. When $\Sigma' = \Sigma$ we simply denote $s^{-1}(\mathcal{L}, \Sigma')$ by $s^{-1}\mathcal{L}$

For a word s and $i \in [1, |w|]$, the i-th letter of s is noted $s_{[i]}$. Given a word s and two integers i, j, s.t. $1 \leq i \leq j \leq |s|$, the *subword* from index i to j is noted $s_{[i \cdots j]}$.

Given two words s and s', we say that s' *is a subsequence of* s, noted $s' \triangleleft s$, if there exists an increasing mapping $k : [1, |s'|] \to [1, |s|]$ (i.e., $\forall i, j \in [1, |s'|] : i < j \implies k(i) < k(j)$) such that $\forall i \in [1, |s'|] : {s'}_{[i]} = s_{[k(i)]}$. Notice that, k being increasing entails that $|s'| \leq |s|$. Intuitively, the image of $[1, |s'|]$ by function k is the set of indexes of letters of s that are "kept" in s'.

Given $\Sigma_1 \subseteq \Sigma$, we define the *projection* operator on finite words, $P_{\Sigma_1} : \Sigma^* \to \Sigma_1^*$, that removes in a word of Σ^* all the events that do not belong to Σ_1. Formally, P_{Σ_1} is recursively defined as follows: $P_{\Sigma_1}(\epsilon) = \epsilon$ and for $\sigma \in \Sigma, u \in \Sigma^*$, $P_{\Sigma_1}(u.\sigma) = P_{\Sigma_1}(u).\sigma$ if $\sigma \in \Sigma_1$ and $P_{\Sigma_1}(u)$ otherwise. Let $K \subseteq \Sigma^*$ be a language. The definition of projection

for words extends to languages: $P_{\Sigma_1}(K) = \{P_{\Sigma_1}(s) \mid s \in K\}$. Conversely, let $K \subseteq \Sigma_1^*$. The *inverse projection* of K is $P_{\Sigma_1}^{-1}(K) = \{s \in \Sigma^* \mid P_{\Sigma_1}(s) \in K\}$.

1.2 Labelled transition system

The model of labelled transition system is commonly used to represent the behavior of systems at a very abstract level. It is composed of a (possibly infinite) number of states (or configurations) and transitions between those states, labeled by actions representing the atomic evolutions of the system.

Definition 1.1 (LTS) *A* labelled transition system $\mathcal{G}$ *is a tuple* $(Q, q_o, \Sigma, \longrightarrow)$ *with* Q *a set of states with a distinguished element* q_o *called* the initial state*,* Σ *is the set of* events *of* $\mathcal{G}$*,* $\longrightarrow \subseteq Q \times \Sigma \times Q$ *is the* partial transition relation*. If* Q *is finite,* $\mathcal{G}$ *is a* finite LTS*.*

This model allows to give a very simple mathematical description of the evolution of systems (control-command system, computer programs, etc).

Example 1.1 *Figure 1.1 is an LTS that models the behavior of a machine that can break down during its normal evolution and further be repaired.*

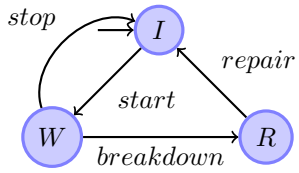


Figure 1.1: Example of LTS

In the rest of the section, we assume given an LTS $\mathcal{G} = (Q, \Sigma, \longrightarrow, q_o)$,

- we write $q \xrightarrow{\sigma} q'$ for $(q, \sigma, q') \in \longrightarrow$. We extend $\longrightarrow$ to arbitrary sequences by setting : $q \xrightarrow{\epsilon} q$ always holds, and $q \xrightarrow{s\sigma} q'$ whenever $q \xrightarrow{s} q''$ and $q'' \xrightarrow{\sigma} q'$, for some $q'' \in Q$.
- Let $q \xrightarrow{s}$ mean that $q \xrightarrow{s} q'$ for some $q' \in Q$. The *event set* of a state $q \in Q$ is $\Sigma(q) \triangleq \{\sigma \in \Sigma \mid q \xrightarrow{\sigma}\}$.
- A state q is *reachable* if $\exists s \in \Sigma^*, q_o \xrightarrow{s} q$.
- We set $\delta_{\mathcal{G}}(q, s) \triangleq \{q' \in Q \mid q \xrightarrow{s} q'\}$. In particular $\delta_{\mathcal{G}}(q, \epsilon) \triangleq \{q\}$. By abuse of notation, for any language $L \subseteq \Sigma^*$,

  $$\delta_{\mathcal{G}}(q, L) \triangleq \{q' \in Q \mid q \xrightarrow{s} q' \text{ for some } s \in L\},$$

  and for any $Q' \subseteq Q$, $\delta_{\mathcal{G}}(Q', L) = \bigcup_{q \in Q'} \delta_{\mathcal{G}}(q, L)$.
- A subset $Q' \subseteq Q$ is *stable* whenever $\delta_{\mathcal{G}}(Q', \Sigma) \subseteq Q'$.

- $\mathcal{G}$ is *alive* if $\Sigma(q) \neq \emptyset$, for each $q \in Q$. It is Σ'*-complete* whenever $\Sigma(q) = \Sigma'$, for each $q \in Q$. It it said to be *complete* whenever it is Σ-complete.
- We say that $\mathcal{G}$ is *deterministic* if whenever $q \xrightarrow{\sigma} q'$ and $q \xrightarrow{\sigma} q''$, then $q' = q''$, for each $q, q', q'' \in Q$ and each $\sigma \in \Sigma$.

A *run* ρ from state q_o in $\mathcal{G}$ is a finite sequence of transitions

$$q_o \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \cdots q_{i-1} \xrightarrow{\sigma_i} q_i \cdots q_{n-1} \xrightarrow{\sigma_n} q_n \tag{1.1}$$

s.t. $\sigma_{i+1} \in \Sigma$ and $q_{i+1} \in \delta_{\mathcal{G}}(q_i, \sigma_{i+1})$ for $i \geq 0$. The *trace* of the run ρ is the word $tr(\rho) = \sigma_1.\sigma_2 \cdots \sigma_n$. We let $last(\rho) = q_n$, and the length of ρ, denoted $|\rho|$, is n. For $i \leq n$ we denote by $\rho(i)$ the prefix of the run ρ truncated at state q_i, i.e., $\rho(i) = q_o \xrightarrow{\sigma_1} q_1 \cdots q_{i-1} \xrightarrow{\sigma_i} q_i$. The set of finite runs from q_o in $\mathcal{G}$ is denoted $Runs(G)$.

A word $u \in \Sigma^*$ is *generated* by $\mathcal{G}$ if $u = tr(\rho)$ for some $\rho \in Runs(\mathcal{G})$. Let $\mathcal{L}(\mathcal{G})$ be the set of words generated by $\mathcal{G}$, which elements are also called *trajectories* of $\mathcal{G}$.

In the next chapters, we will often need to distinguish a subset $F \subseteq Q$ to denote final states. Such states can be used to encode the achievement of a task or configurations of the system that violate some properties, etc. The word $s \in \Sigma^*$ is *accepted* by $\mathcal{G}$ if $s = tr(\rho)$ for some $\rho \in Runs(\mathcal{G})$ with $last(\rho) \in F$. The *language of (finite) words* $\mathcal{L}_F(\mathcal{G})$ of $\mathcal{G}$ is the set of words accepted by $\mathcal{G}$, i.e., $\mathcal{L}_F(\mathcal{G}) = \{s \in \Sigma^* \mid \delta_{\mathcal{G}}(q_o, s) \subseteq F\}$. Note that an LTS equipped with a set of final states F is usually called in the literature an *automaton*.

Parallel Composition of LTSs. In several situations, a system is initially given by a collection of components modeled by LTSs that interact with each other by sharing common events. The global behavior of this system is then obtained by composing these LTSs together using the *parallel composition* operator that represents the concurrent behavior of the LTS with synchronization on the common events.

Definition 1.2 *Let* $\mathcal{G}_1 = (Q_1, q_o^1, \Sigma_1, \longrightarrow_1)$ *and* $\mathcal{G}_2 = (Q_2, q_o^2, \Sigma_2, \longrightarrow_2)$. *The* parallel composition *of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *is the LTS* $\mathcal{G}_1 \times \mathcal{G}_2 = (Q, q_o, \Sigma, \longrightarrow)$ *where*

- $Q = Q_1 \times Q_2$, $q_o = (q_o^1 \times q_o^2)$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\longrightarrow$ *is the smallest relation in* $Q \times \Sigma \times Q$ *satisfying*

$$(q_1, q_2) \xrightarrow{\sigma} \begin{cases} (q_1', q_2') \text{ if } \sigma \in \Sigma_1 \cap \Sigma_2 \text{ and } q_i \xrightarrow{\sigma}_i q_i', i = 1, 2 \\ (q_1', q_2) \text{ if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } q_1 \xrightarrow{\sigma}_1 q_1' \\ (q_1, q_2') \text{ if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } q_2 \xrightarrow{\sigma}_2 q_2' \end{cases} \tag{1.2}$$

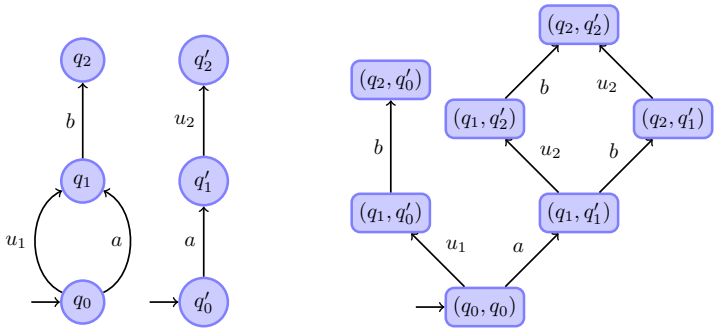
Using the projection $P_{\Sigma_i} : \Sigma^* \to \Sigma_i^*$, $i = 1, 2$, we can characterize the languages resulting from the parallel composition as follows:

$$\mathcal{L}(\mathcal{G}_1 \times \mathcal{G}_2) = P_{\Sigma_1}^{-1}(\mathcal{L}(\mathcal{G}_1)) \cap P_{\Sigma_2}^{-1}(\mathcal{L}(\mathcal{G}_2) \tag{1.3}$$

Moreover, if the two LTSs are equipped with a set of final states (F_1 and F_2), then we have:

$$\mathcal{L}_{F_1 \times F_2}(\mathcal{G}_1 \times \mathcal{G}_2) = P_{\Sigma_1}^{-1}(\mathcal{L}_{F_1}(\mathcal{G}_1)) \cap P_{\Sigma_2}^{-1}(\mathcal{L}_{F_2}(\mathcal{G}_2) \tag{1.4}$$

Example 1.2 *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two LTSs over respectively $\Sigma_1 = \{a, u_1, b\}$ and $\Sigma_2 = \{a, u_2\}$ represented in Fig. 1.2(a). The LTS $\mathcal{G}_1 \times \mathcal{G}_2$ obtained by composing these two LTS is depicted in Fig. 1.2(b).*



(a) $\mathcal{G}_1$ and $\mathcal{G}_2$ (b) $\mathcal{G}_1 \times \mathcal{G}_2$

Figure 1.2: parallel composition of two LTS

State predicates. Given a set of states $E \subseteq Q$ of an LTS $\mathcal{G}$, the functions $Post_{\mathrm{G}}$, $Pre^{\forall}_{\mathrm{G}}$ and $Pre^{\exists}_{\mathrm{G}}$ from $2^Q \to 2^Q$ are defined as follows:

$$Post_{\mathcal{G}}(E) = \{\delta_{\mathrm{G}}(q, \sigma) \mid \exists \sigma \in \Sigma, q \in E\} \tag{1.5}$$

$$Pre^{\exists}_{\mathcal{G}}(E) = \{q \in Q \mid \exists \sigma \in \Sigma, \delta_{\mathrm{G}}(q, \sigma) \cap E \neq \emptyset\} \tag{1.6}$$

$$Pre^{\forall}_{\mathcal{G}}(E) = \{q \in Pre^{\exists}_{\mathrm{G}}(E) \mid \forall \sigma \in \Sigma, \delta_{\mathrm{G}}(q, \sigma) \subseteq E\} \tag{1.7}$$

The states of $Post_{\mathrm{G}}(E)$ are the immediate successors of E in $\mathcal{G}$. The states belonging to $Pre^{\forall}_{\mathrm{G}}(E)$ are the states such that all immediate successors belong to E, while the states belonging to $Pre^{\exists}_{\mathrm{G}}(E)$ are such that at least one immediate successor belongs to E.

Given a live LTS $\mathcal{G}$, let $Reach_{\mathrm{G}}(E)$ be the set of states that can be reached from E, $Inev_{\mathrm{G}}(E)$ be the set of states that inevitably lead to a set E in a finite number of steps and $CoReach_{\mathrm{G}}(E)$ the set of states from which E is reachable. These sets are given by the following least fix-points (lfp):

$$Reach_{\mathrm{G}}(E) = lfp(\lambda X.E \cup Post^{\forall}_{\mathcal{G}}(X)) \tag{1.8}$$

$$Inev_{\mathrm{G}}(E) = lfp(\lambda X.E \cup Pre^{\forall}_{\mathcal{G}}(X)) \tag{1.9}$$

$$CoReach_{\mathrm{G}}(E) = lfp(\lambda X.E \cup Pre^{\exists}_{\mathcal{G}}(X)) \tag{1.10}$$

Note that by the Tarski's theorem [Tar55], since the functions $E \cup post_{\mathrm{G}}(X)$, $E \cup Pre^{\forall}_{\mathrm{G}}(X)$ and $E \cup Pre^{\exists}_{\mathrm{G}}(X)$ are monotonic, the limit of the previous fix-points actually exists (but may be uncomputable as the state space is possibly infinite).

Observable behavior. In the next chapters, partial observation will play a central role. In this regard, the set of events Σ can partitioned into Σ_o and Σ_{uo}

$$\Sigma = \Sigma_o \cup \Sigma_{uo}, \text{ and } \Sigma_o \cap \Sigma_{uo} = \emptyset,$$

where Σ_o represents the set of *observable* events. Elements of Σ_{uo} are *unobservable* events. Typical elements of Σ_o^* will be denoted by μ, μ'. We say that $\mathcal{G}$ is Σ_o-*alive* if $\forall q \in Q, \exists s \in \Sigma^*.\Sigma_o, q \xrightarrow{s}$, meaning that there is no terminal loop of unobservable events. Notice that when $\mathcal{G}$ has no loop of unobservable events, $\mathcal{G}$ is alive if and only if $\mathcal{G}$ is Σ_o-alive. Now, the *language of traces of* $\mathcal{G}$ is given by

$$\mathit{Traces}_{\Sigma_o}(\mathcal{G}) \triangleq P_{\Sigma_o}(\mathcal{L}(\mathcal{G}))$$

It is the set of observable sequences of its trajectories.

Next, we introduce the unobservable-closure $\textsc{Uc}(\mathcal{G})$ of an LTS $\mathcal{G}$, in order to abstract out unobservable events according to the projection P_{Σ_o}.

Definition 1.3 *For an LTS* $\mathcal{G} = (Q, \Sigma, \longrightarrow, q_o)$, *the* unobservable-closure of $\mathcal{G}$ *is the LTS* $\textsc{Uc}(\mathcal{G}) = (Q, \Sigma_o, \longrightarrow_o, q_o)$ *where for any* $q, q' \in Q, \sigma \in \Sigma_o$, $q \xrightarrow{\sigma}_o q'$ *in* $\textsc{Uc}(\mathcal{G})$ *whenever there exists* $s \in \Sigma_{uo}^*$ *such that* $q \xrightarrow{s\sigma} q'$ *in* $\mathcal{G}$. $\diamond$

We get $\mathcal{L}(\textsc{Uc}(\mathcal{G})) = P_{\Sigma_o}(\mathcal{L}(\mathcal{G}))$ and for $F \subseteq Q$, $\mathcal{L}_F(\textsc{Uc}(\mathcal{G})) = P_{\Sigma_o}(\mathcal{L}_F(\mathcal{G}))$.

From the projection P_{Σ_o}, we derive an equivalence relation between trajectories of $\mathcal{G}$, written $\sim_{\mathcal{G}}$, called the *delay-observation equivalence* in reference to the delay-bisimulation of [Mil81]:

Definition 1.4 (Delay-Observation Equivalence, $\sim_{\mathcal{G}}$) *Let* $\sim_{\mathcal{G}} \subseteq \mathcal{L}(\mathcal{G}) \times \mathcal{L}(\mathcal{G})$ *be the binary relation defined by* $s \sim_{\mathcal{G}} s'$ *whenever*

- $P_{\Sigma_o}(s) = P_{\Sigma_o}(s')$ *and*
- $s \in \Sigma^*\Sigma_o$ *if and only if* $s' \in \Sigma^*\Sigma_o$.

One easily verifies that $\sim_{\mathcal{G}}$ *is an equivalence relation, and we write* $[s]$ *for the equivalence class of* s.

Given $s \in \mathcal{L}(\mathcal{G})$, s naturally maps onto a trace of $\mathcal{G}$, namely $P_{\Sigma_o}(s)$. Now, given a non empty trace μ of $\mathcal{G}$, μ does not uniquely determine a delay-observation equivalence class as in general μ can be brought back in $\mathcal{G}$ in at least two different manners:

- μ can be associated with the class $[s]$ with $P_{\Sigma_o}(s) = \mu$ and $s \in \Sigma^*\Sigma_o$,
- or μ can be associated with the class $[s']$ with $P_{\Sigma_o}(s') = \mu$ and $s' \in \Sigma^*\Sigma_{uo}$

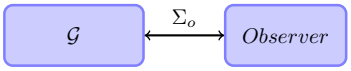
Notice that by Definition 1.4, $[s]$ and $[s']$ are different. Henceforth, the equivalence class denoted by a trace μ is

$$[\![\mu]\!]_{\Sigma_o} \triangleq \begin{array}{l} P_{\Sigma_o}^{-1}(\mu) \cap \mathcal{L}(\mathcal{G}) \cap \Sigma^*\Sigma_o \text{ if } \mu \neq \epsilon \\ [\epsilon] \text{ otherwise.} \end{array} \tag{1.11}$$

We say that $[\![\mu]\!]_{\Sigma_o}$ is the set of trajectories *compatible* with the trace μ. When clear from the context, we will use $[\![\mu]\!]$ for $[\![\mu]\!]_{\Sigma_o}$. The above semantics is consistent with an on-line observation performed by a user of the system for whom the system is only seen through the interface given by the observation mask P_{Σ_o}. We suppose that users are reacting faster

than the system. Therefore, when an observable event occurs, a user can take a decision before the system proceeds with any unobservable event. This explains why we do not consider trajectories ending with unobservable events in the definition of the semantics.

Next, we might be interested by the possible states in which the system can be, based on the observation (it might be due to the fact that some events are unobservable or because $\mathcal{G}$ is non-deterministic.



This is formalized by the notion of determinisation:

Definition 1.5 *Let $\mathcal{G} = (Q, \Sigma, \longrightarrow, q_0)$ be an LTS with $\Sigma = \Sigma_{uo} \cup \Sigma_o$. The* determinization *of $\mathcal{G}$ is the LTS $Det_{\Sigma_o}(\mathcal{G}) = (\mathcal{X}, \Sigma_o, \rightarrow_d, X_0)$ where $\mathcal{X} = 2^Q$ (the set of subsets of Q called* macro-states*), $X_0 = \{q_0\}$ and $\rightarrow_d = \{(X, \sigma, \delta_{\mathcal{G}}(X, \Sigma_{uo}^*.\sigma)) \mid X \in \mathcal{X} \text{ and } \sigma \in \Sigma_o\}$.*

Notice that for this definition the target macro-state X' of a transition $X \xrightarrow{\sigma}_d X'$ is only composed of states q' of $\mathcal{G}$ which are targets of sequences of transitions $q \xrightarrow{s.\sigma} q'$ ending with an observable event σ. The reason for this definition is the coherency with $[\![.]\!]_{\Sigma_o}$. In fact, from the definition of $\rightarrow_d$ in $Det_{\Sigma_o}(\mathcal{G})$, we infer that $\delta_{Det_{\Sigma_o}(\mathcal{G})}(X_0, \mu) = \{\delta_{\mathcal{G}}(q_0, [\![\mu]\!]_{\Sigma_o})\}$, which means that the macro-state reached from X_0 by μ in $Det_{\Sigma_o}(\mathcal{G})$ is composed of the set of states that are reached from q_0 by trajectories of $[\![\mu]\!]_{\Sigma_o}$ in $\mathcal{G}$.

Finally, determinisation preserves traces, so we have $\mathcal{L}(Det_{\Sigma_o}(\mathcal{G})) = Traces_{\Sigma_o}(Det_{\Sigma_o}(\mathcal{G})) = Traces_{\Sigma_o}(\mathcal{G})$.

Example 1.3 *In order to illustrate the previous definition, let us consider the following LTS G (left-hand side of Fig. 1.3). The corresponding $Det_{\Sigma_o}(\mathcal{G})$ is depicted on the right-hand side of Fig. 1.3.*

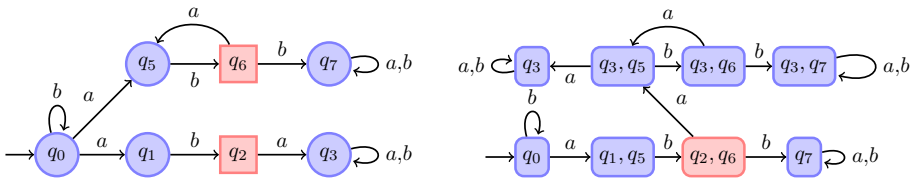


Figure 1.3: Example of determinisation

Chapter 2

Diagnosis of Discrete Event Systems

Besides model-checking and conformance model testing that are traditionally used to verify the properties that hold on the model on the system, one can be interested in identifying on-line that some particular behaviours (called faults) occur on the actual system $\mathcal{I}$ that is described by a formal model $\mathcal{G}$ of the system (note that $\mathcal{G}$ is supposed to be correct w.r.t. $\mathcal{I}$. Diagnosing and monitoring dynamical systems is an increasingly active research domain and various model-based approaches have been proposed which differ according to the kind of models they used [SSL+96, BLPZ98, RC98, PC05, DLT00a, FBJ+00]. Most of the time, fault diagnosis aims at achieving the detection of the faults. The challenge in diagnosis is hence to build accurate diagnosers, that do not miss faults, but do not raise false alarms either. A diagnoser can be seen as a state estimator for the monitored system, that builds from an observation the set of possible current states of the system. When the estimation contains only faulty states, a diagnoser can claim that a fault occurred. Conversely, when the estimation contains only non-faulty states, the diagnoser can claim that the system is in a safe state.

In this chapter, we are interested in the fault detection that might arise in a system. In the sequel, we shall consider the framework introduced by [SSL+96], in which the faulty behaviour is included in the system modeled by an LTS. In this setting, the general diagnosis problem is thus to detect or identify patterns of particular events (i.e., a particular behaviour modeling a faulty behaviour) on a partially observable system. The aim of diagnosis is to decide, by means of a *diagnoser*, whether or not such a pattern occurred in the system. Even if such a decision cannot be taken immediately after the occurrence of the pattern, one requires that this decision has to be taken in a bounded delay. This property is usually called *diagnosability*. This property can be checked *a priori* from the system model, and depends on its observability and on the kind of faults which are looked for.

The diagnosis problem has been tackled with different formalisms from the LTS [SSL+96] model and their timed or probabilistic extension, Petri Nets, High-level Message Sequence Charts [J7], recursive tile systems, which are recursive infinite systems generated by a finite collection of finite tiles, a simplified variant of deterministic graph

grammars (slightly more general than push-down systems) [J4]. As stressed earlier in order to solve the diagnosis problem, one has to assume that the faulty behaviour of the system is known. Faulty models depend on the kind of faults we are looking for: it might be the execution of an event, reaching a faulty state (state-based diagnosis), the execution of a supervision pattern or of a particular sequence of a language. Most of the times, the faulty behaviour is considered as stable, meaning that once the fault occurred, the system remains faulty forever. Meanwhile, the problem of diagnosing intermittent faults has been tackled in [CLT04]. Finally, following the supervisory control problem, different kinds of architecture of the diagnoser can be considered: the diagnoser can be centralized (it observes the systems through a particular interface (a subset of the possible events or a mask) and takes its decision according to what it observes. The diagnoser can be decentralized and in that case, each local diagnoser has a particular view of the system, takes its own decision regarding the presence of fault and a global oracle "compiles" the decisions of all the diagnosers in order to decide whether a fault occurred or not in the system. We refer to [ZL13] for a complete review of the diagnosis problem depending on the different kinds of models of systems or faults that are considered.
The contents of this chapter is as follows:

- In Section 2.1, we propose a model of supervision patterns general enough to capture past occurrences of particular trajectories of the system. Modeling the diagnosis objective by a supervision pattern allows us to generalize the properties to be diagnosed and to render them independent of the description of the system [C22].

- In Section 2.2, we are going one step ahead and [C22] is extended for the prediction of supervision patterns. The occurrences of the pattern are predictable if it is possible to infer about any occurrence of the pattern before the pattern is completely executed by the system. We designed an off-line algorithm to verify the property of predictability. We have shown that the verification is polynomial in the number of states of the system. We further designed an on-line algorithm to track the execution of the pattern during the operation of the system. This algorithm is based on the use of a diagnoser LTS [C19].

- In Section 2.3, we examine the case of transient faults, that can appear and be repaired. Diagnosability in this setting means that the occurrence of a fault should always be detected in bounded time, but also before the fault is repaired. Checking this notion of diagnosability is proved to be PSPACE-complete. It is also shown that faults can be reliably counted provided the system is diagnosable for faults and for repairs [C1].

2.1 Diagnosis of Stable Supervision Patterns

Even-though well defined by [SSL+96] and others, one observes many different definitions of diagnosability and ad-hoc algorithms for the construction of the diagnoser, as well as for the verification of diagnosability. As a consequence, all these results are difficult to reuse for new but similar diagnosis problems. We believe that the reason comes from an absence of a clear definition of the involved patterns, which would clarify the separation between the diagnosis objective and the specification of the system. In this section, we formally introduce the notion of supervision pattern as a means to define the diagnosis objectives: a supervision pattern is an automaton which language is the set of trajectories one wants to diagnose. The proposal is general enough to cover in an unified way an important class of diagnosis objectives, including detection of permanent faults, but also transient faults, multiple faults, repeating faults, as well as quite complex sequences of events. We then propose a formal definition of the Diagnosis Problem in this context. The essential point is a clear definition of the set of trajectories *compatible* with an observed trace. Now, the Diagnosis Problem is expressed as the problem of synthesizing a function over traces, the *diagnoser*, which decrees on the possible/certain occurrence of the pattern on trajectories compatible with the trace. The diagnoser is required to fulfil two fundamental properties: *correctness* and *bounded diagnosability*. *Correctness* expresses that the diagnoser answers accurately and *Bounded Diagnosability* guarantees that only a bounded number of observations is needed to eventually answer with certainty that the pattern has occurred. *Bounded Diagnosability* is formally defined as the Ω-diagnosability of the system (where Ω is the supervision pattern), which compares to standard diagnosability by [SSL+96]. Relying on the formal framework we have developed, we then propose algorithms for both the diagnoser's synthesis, and the verification of Ω-diagnosability. We believe that these generic algorithms as well as their correctness proofs are a lot more simple than the ones proposed in the literature.

2.1.1 Supervision Patterns

In this section, we introduce the notion of *supervision patterns*, which are means to define languages we are interested in for diagnosis purpose. We then give some examples of such patterns. Finally, we introduce the diagnosis problem for such patterns. Supervision patterns are represented by particular LTSs:

Definition 2.1 *A* supervision pattern *is a 5-tuple* $\Omega = (Q_\Omega, \Sigma, \longrightarrow_\Omega, q_{0_\Omega}, Q_F)$*, where* $(Q_\Omega, \Sigma, \longrightarrow_\Omega, q_{0_\Omega})$ *is a deterministic and complete LTS, and* $Q_F \subseteq Q \setminus \{q_{0_\Omega}\}$ *is a distinguished stable*[1] *subset of states.*

As Ω is complete we get $\mathcal{L}(\Omega) = \Sigma^*$. Also notice that the assumption that Q_F is stable means that its accepted language is "extension-closed", i.e. satisfies $\mathcal{L}_{Q_F}(\Omega).\Sigma^* = \mathcal{L}_{Q_F}(\Omega)$. Otherwise said, $\mathcal{L}_{Q_F}(\Omega)$ is a complement of a safety property. This choice is natural since we want to diagnose whether all trajectories compatible with an observed trace have a prefix recognized by the pattern. Notice that Definition 2.1 requires that $q_{0_\Omega} \notin Q_F$; this is very natural. Indeed, a supervision pattern is not designed to declare a fault *a priori*

[1] By stable, we mean that all transitions starting from a state in Q_F reach a state also in Q_F.

while the system has not made any transition yet. Henceforth, we do not consider that unexpected behaviour (membership in Q_F) can occur from the very initial state of a system.

We now give some examples of supervision patterns which rephrase classical properties one is interested in for diagnosis purpose.

Occurrence of one fault. Le $f \in \Sigma$ be a fault and consider that we are interested in diagnosing the occurrence of this fault. A trajectory $s \in \Sigma^*$ is faulty if $s \in \Sigma^*.f.\Sigma^*$. The supervision pattern Ω_f of Figure 2.1 exactly recognizes this language, $\mathcal{L}_F(\Omega_f) = \Sigma^*.f.\Sigma^*$.

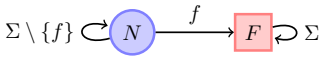


Figure 2.1: Supervision pattern for one fault

The aim of this pattern is to mimic the diagnosis of the occurrence of a fault as introduced in [SSL+96].

Occurrence of multiple faults. Let f_1 and f_2 be two faults that may occur in the system. Diagnosing the occurrence of these two faults in a trajectory means deciding the membership of this trajectory in $\Sigma^*.f_1.\Sigma^* \cap \Sigma^*.f_2.\Sigma^* = \mathcal{L}_{F_1}(\Omega_{f_1}) \cap \mathcal{L}_{F_2}(\Omega_{f_2})$, where $\Omega_{f_i}, i \in \{1, 2\}$ are isomorphic to the supervision pattern Ω_f described in Figure 2.1. The supervision pattern is then the product $\Omega_{f_1} \times \Omega_{f_2}$ which accepted language in $F_1 \times F_2$ is $\mathcal{L}_{F_1 \times F_2}(\Omega_{f_1} \times \Omega_{f_2}) = \mathcal{L}_{F_1}(\Omega_{f_1}) \cap \mathcal{L}_{F_2}(\Omega_{f_2})$.

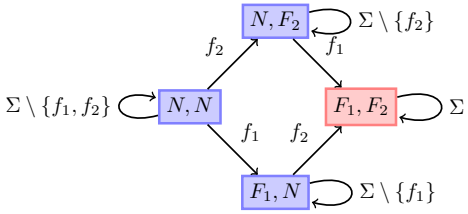


Figure 2.2: Supervision pattern for two faults

More generally, the supervision pattern for the occurrence of a set of faults $\{f_1, \cdots, f_l\}$ is the product $\times_{i=1,\ldots,l}\Omega_{f_i}$, considering $\times_{i=1,\ldots,l}F_i$ as final state set.

Ordered occurrence of events. If the diagnosis that has to be performed concerns the occurrences of different faults in a precise order, for example, f_2 after f_1, the trajectories that have to be recognized by the supervision pattern are $\Sigma \setminus \{f_1\}^*.f_1.\Sigma \setminus \{f_2\}^*.f_2.\Sigma^*$. If f_1 corresponds to a fault event and f_2 to the reparation of this fault in the system, then we actually diagnose the reparation of the fault f_1. With this pattern, the aim is to match the *I-diagnosability* in [CLT04].

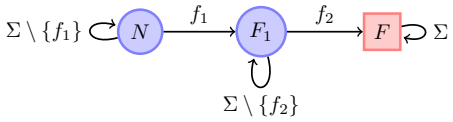


Figure 2.3: Ordered occurrence of events

Multiple occurrences of the same fault. Another interesting problem is to diagnose the multiple occurrences of the same fault event f, say k times. The supervision pattern is given in Figure 2.4 which accepted language is $\mathcal{L}_F(\Omega_f)^k$. The aim is to match the k-diagnosability of [JKG03].

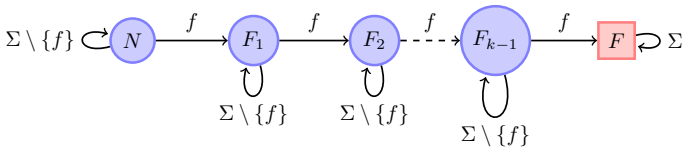


Figure 2.4: k occurrences of the same fault f

This can be easily generalized to a pattern recognizing the occurrence of k patterns (identical or not). One can also consider as faulty set of final states the sets $(\{F_i, \cdots, F_{k-1}, F\})_{k' \leq i \leq k}$. A diagnoser computed with respect to one of these sets diagnoses that the fault f occurs at least i times (with $i \geq k'$). Now we can also consider the union of the verdicts of all these diagnosers. We thus diagnose that a fault occurred at least between k' and k times. With $k' = 1$, this actually corresponds to the $[1-k]$-diagnosability of [JKG03].

Intermittent Fault. So far, we have considered permanent faults. However, there exist numerous systems in which faults are intermittent (i.e. the effect of the fault can be repaired). We here assume that reparation of a fault f is encoded by means of an event r (see [CLT04] for details). We shall come back to this aspect in the next section for particular kinds of intermittent faults that cannot be represented by a stable supervision pattern.

The supervision pattern given in Figure 2.5 describes the fact that a fault (occurrence of f) occurred twice without repair (occurrence of r).

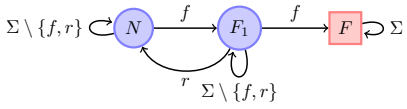


Figure 2.5: Intermittent fault with repair

It is worthwhile noting that this can be generalized to a pattern recognizing the occurrence of k faults (identical or not) without repair.

2.1.2 The Diagnosis Problem

In the remainder of the paper, we consider a system whose behavior is modeled by an LTS $\mathcal{G} = (Q, \Sigma, \longrightarrow, q_o)$. The only assumption made on $\mathcal{G}$ is that $\mathcal{G}$ is Σ_o-alive. Notice that $\mathcal{G}$ can be non-deterministic. We also consider a supervision pattern $\Omega = (Q_\Omega, \Sigma, \longrightarrow_\Omega , q_{o_\Omega}, Q_F)$ denoting the language $\mathcal{L}_{Q_F}(\Omega)$ that we want to diagnose.

We define the *Diagnosis Problem* as the problem of defining a function $\texttt{Diag}_\Omega$ on traces whose intention is to answer the question whether trajectories corresponding to observed traces are recognized or not by the supervision pattern. We do require some properties for $\texttt{Diag}_\Omega$: *Correctness* and *Bounded Diagnosability*.

- *Correctness* means that "Yes" and "No" answers should be accurate
- *Bounded Diagnosability* means that trajectories in $\mathcal{L}_{Q_F}(\Omega)$ should be diagnosed with finitely many observations.

The *Diagnosis problem* can be stated as follows: given an LTS $\mathcal{G}$ and given a supervision pattern Ω, decide whether there exists (and compute if any) a three valued function $\texttt{Diag}_\Omega : Traces(\mathcal{G}) \to \{\text{"YES"}, \text{"NO"}, \text{"?"}\}$ decreeing, for each trace μ of $\mathcal{G}$, on the membership in $\mathcal{L}_{Q_F}(\Omega)$ of any trajectory in $[\![\mu]\!]_{\Sigma_o}$. Formally,

- (Diagnosis Correctness) The function should verify
$$\texttt{Diag}_\Omega(\mu) \quad = \begin{cases} \text{"YES"} & \text{if } [\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_{Q_F}(\Omega) \\ \text{"NO"} & \text{if } [\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_{Q_F}(\Omega) = \emptyset \\ \text{"?"} & \text{otherwise.} \end{cases}$$
- (Bounded Diagnosability) As $\mathcal{G}$ is only partially observed, we expect in general situations where $\texttt{Diag}_\Omega(\mu) = \text{"?"}$ (as neither $[\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_{Q_F}(\Omega)$ nor $[\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_{Q_F}(\Omega) = \emptyset$ hold). However, we require this undetermined situation not to last in the following sense: there must exist $n \in \mathbb{N}$, the bound, such that whenever $s \in [\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_{Q_F}(\Omega)$, for all $t \in \mathcal{L}(\mathcal{G})/s \cap \Sigma^*.\Sigma_o$, if $|P_{\Sigma_o}(t)| \geq n$ then $\texttt{Diag}_\Omega(P_{\Sigma_o}(s.t)) = \text{"YES"}$.

Diagnosis Correctness means that the diagnosis of a trace μ is "No" if no trajectory in its semantics $[\![\mu]\!]_{\Sigma_o}$ lies in $\mathcal{L}_{Q_F}(\Omega)$ while it is "Yes" if all trajectories in $[\![\mu]\!]_{\Sigma_o}$ lie in $\mathcal{L}_{Q_F}(\Omega)$. Bounded Diagnosability means that when observing a trajectory in $\mathcal{L}_{Q_F}(\Omega)$, a "Yes" answer should be produced after finitely many observable events (See 2.6 for an intuitive explanation of these notions).
Now, if $\texttt{Diag}_\Omega$ provides a Correct Diagnosis, Bounded Diagnosability can be rephrased, by replacing $\texttt{Diag}_\Omega(P_{\Sigma_o}(s.t)) = \text{"YES"}$ with $[\![P_{\Sigma_o}(s.t)]\!]_{\Sigma_o} \subseteq \mathcal{L}_{Q_F}(\Omega)$. We obtain what we call the Ω-*diagnosability*. Notice that this is now a property of $\mathcal{G}$ with respect to Ω.

Definition 2.2 *An LTS $\mathcal{G}$ is* $\Omega(n)$-diagnosable, *where $n \in \mathbb{N}$, whenever*

$$\begin{aligned} \forall s \in \mathcal{L}_{Q_F}(\Omega) \cap \mathcal{L}(\mathcal{G}) \cap \Sigma^*\Sigma_o, \forall t \in \mathcal{L}(\mathcal{G})/s \cap \Sigma^*\Sigma_o, \\ \textit{if } |P_{\Sigma_o}(t)| \geq n \ \textit{ then } [\![P_{\Sigma_o}(s.t)]\!]_{\Sigma_o} \subseteq \mathcal{L}_{Q_F}(\Omega), \end{aligned} \tag{2.1}$$

We say that $\mathcal{G}$ is Ω-diagnosable *if $\mathcal{G}$ is* $\Omega(n)$-diagnosable *for some $n \in \mathbb{N}$.*

Ω-diagnosability says that when a trajectory s ending with an observable event is recognized by the supervision pattern Ω, for any extension t with enough observable events, any

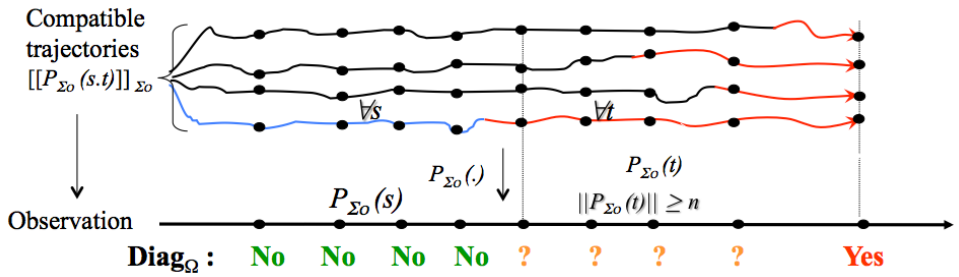


Figure 2.6: the Ω-diagnosability for $\Omega = \Omega_f$

trajectory s' compatible with the observation $P_{\Sigma_o}(s.t)$ is also recognized by Ω. The remark before Definition 2.2 is formalized by :

Proposition 2.1 *If* $\texttt{Diag}_\Omega$ *computes a Correct Diagnosis, then* $\mathcal{G}$ *is* Ω*-diagnosable if and only if the Bounded Diagnosability Property holds for* $\texttt{Diag}_\Omega$.

As to show the unifying framework based on supervision patterns, we here consider the very particular supervision pattern Ω_f of Section 2.1.1, originally considered by [SSL+95, SSL+96] with the associated notion of f-diagnosability. Let us first recall this notion. Let $\mathcal{G}$ be an LTS which is alive and has no loop of unobservable event. $\mathcal{G}$ is *f-diagnosable* whenever

$$\begin{array}{c} \exists N \in \mathbb{N}, \forall s \in \Sigma^*.f, \forall t \in \mathcal{L}(\mathcal{G})/s, \text{ if } |t| \geq N, \\ \text{then } \forall u \in \mathcal{L}(\mathcal{G}), u \sim_{\Sigma_0} s.t \Rightarrow u \in \Sigma^*.f.\Sigma^* \end{array} \tag{2.2}$$

For the f-diagnosability to guarantee that the Bounded Diagnosability Property is achievable, it is necessary to assume that each infinite trajectory in $\mathcal{G}$ contains at least one observable event. The following proposition relates f-diagnosability with Ω_f-diagnosability:

Proposition 2.2 *Let* $\mathcal{G}$ *be an LTS and assume that* $\mathcal{G}$ *is alive and has no loop of internal events. Then* $\mathcal{G}$ *is* f*-diagnosable if and only if* $\mathcal{G}$ *is* Ω_f*-diagnosable.*

2.1.3 Algorithms for the Diagnosis Problem

We now propose algorithms for the Diagnosis Problem based on standard operations on LTSs. In a first stage we base the construction of the $\texttt{Diag}_\Omega$ function on the synchronous product of $\mathcal{G}$ and Ω and its determinisation, and prove that the function $\texttt{Diag}_\Omega$ computes a Correct Diagnosis. Next, we propose an algorithm allowing to check for the Ω-diagnosability of an LTS, thus ensuring the Bounded Diagnosability Property of the function $\texttt{Diag}_\Omega$. Hence achieving the decision of the Diagnosis Problem.

2.1.3.1 Computing a candidate for the function $\texttt{Diag}_\Omega$

We propose a computation of the function $\texttt{Diag}_\Omega$: given $\mathcal{G}$ an LTS and a supervision pattern Ω, we first consider the synchronous product $\mathcal{G}_\Omega$ of $\mathcal{G}$ and Ω (see Definition 1.2). Next

we perform on $\mathcal{G}_\Omega$ a second operation (see Definition 1.5) which associates to $\mathcal{G}_\Omega$ a deterministic LTS written $Det_{\Sigma_o}(\mathcal{G}_\Omega)$. We then show how $Det_{\Sigma_o}(\mathcal{G}_\Omega)$ provides a function $\texttt{Diag}_\Omega$ delivering a Correct Diagnosis.

We now explain the construction of the diagnoser from $\mathcal{G}$ and Ω. Let us first consider the synchronous product $\mathcal{G}_\Omega = \mathcal{G} \times \Omega$ (see Definition 1.2). We then get $\mathcal{L}(\mathcal{G}_\Omega) = \mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\Omega) = \mathcal{L}(\mathcal{G})$ as Ω is complete (thus $\mathcal{L}(\Omega) = \Sigma^*$). We also get $\mathcal{L}(\mathcal{G}_\Omega, Q \times Q_F) = \mathcal{L}(\mathcal{G}) \cap \mathcal{L}_{Q_F}(\Omega)$ meaning that the trajectories of $\mathcal{G}$ accepted by Ω are exactly the accepted trajectories of $\mathcal{G}_\Omega$. Finally, note that $Q \times Q_F$ is stable in $\mathcal{G}_\Omega$ as both Q and Q_F are stable by assumption.

We now apply determinisation to $\mathcal{G}_\Omega$. We have $\mathit{Traces}_{\Sigma_o}(Det_{\Sigma_o}(\mathcal{G}_\Omega)) = \mathit{Traces}_{\Sigma_o}(\mathcal{G}_\Omega) = Traces_{\Sigma_o}(\mathcal{G})$ thus for all $\mu \in Traces_{\Sigma_o}(\mathcal{G})$, $\delta_{Det_{\Sigma_o}(\Sigma_o\mathcal{G}_\Omega)}(X_o, \mu) = \{\delta_{\mathcal{G}_\Omega}(q_o, [\![\mu]\!]_{\Sigma_o}\}$.

We now establish the following fundamental results on the construction $Det_{\Sigma_o}(\mathcal{G}_\Omega)$:

Proposition 2.3 *For any $\mu \in Traces_{\Sigma_o}(\mathcal{G}) = Traces_{\Sigma_o}(\mathcal{G}_\Omega)$,*

$$\delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X_o, \mu) \subseteq Q \times Q_F \iff [\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_{Q_F}(\Omega) \tag{2.3}$$

$$\delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X_o, \mu) \cap Q \times Q_F = \emptyset \iff [\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_{Q_F}(\Omega) = \emptyset \tag{2.4}$$

(2.3) means that all trajectories compatible with a trace μ are accepted by Ω if and only if μ leads to a macro-state only composed of marked states in $\mathcal{G}_\Omega$. (2.4) means that all trajectories compatible with μ are not accepted by Ω if and only if μ leads to a macro-state only composed of unmarked states in $\mathcal{G}_\Omega$.

We have now the material to define the function $\texttt{Diag}_\Omega$ and to obtain the Diagnosis Correctness Property, following directly from Proposition 2.3.

Theorem 2.1 *Let $Det_{\Sigma_o}(\mathcal{G}_\Omega)$ be the LTS built as above, and let $\texttt{Diag}_\Omega(\mu)$ be:*

$$\begin{cases} \text{“YES”, if} & \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X_0, \mu) \subseteq Q \times Q_F \\ \text{“NO”, if} & \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X_0, \mu) \cap Q \times Q_F = \emptyset \\ \text{“?”, otherwise.} & \end{cases} \tag{2.5}$$

$\texttt{Diag}_\Omega$ computes a Correct Diagnosis.

Example 2.1 *In order to illustrate the diagnoser construction, consider the LTS $\mathcal{G}$ of Fig. 2.7 (left-hand side). Assume we want to diagnose the occurrence of the fault event f. We thus use the supervision pattern Ω_f described in Figure 2.1 and build the product $\mathcal{G}_\Omega = G \times \Omega_f$. In this case $\mathcal{G}_\Omega$ is isomorphic to $\mathcal{G}$ with set of marked states $\{q_1, q_2, q_3\}$. The diagnoser (as well as its answers) obtained by determinization of $\mathcal{G}_\Omega$ is also represented in Figure 2.7 (right-hand side). The red square means that the diagnoser emits the "YES" verdict, the dashed-line borders rectangle means the diagnoser emits the "?" verdict whereas the "NO" verdict is emitted in the other cases.* ◇

2.1.3.2 Verifying the Bounded Diagnosability Property of $\texttt{Diag}_\Omega$

As we have established the Correctness of $\texttt{Diag}_\Omega$, according to Proposition 2.1, the Bounded Diagnosability Property of $\texttt{Diag}_\Omega$ is provided by the Ω-diagnosability of $\mathcal{G}$. We now propose an algorithm for deciding Ω-diagnosability (Definition 2.2).

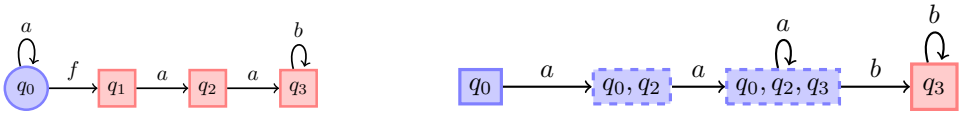
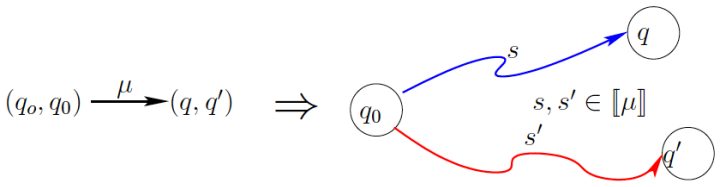


Figure 2.7: $\mathcal{G}_\Omega$ and its associated diagnoser computed w.r.t. Ω_f

This algorithm is adapted from [JHCK01, YL02]. The idea is that $\mathcal{G}$ is not Ω-diagnosable if there exists an arbitrarily long trace μ, such that two trajectories compatible with μ disagree on membership to $\mathcal{L}_{Q_F}(\Omega)$ (see the above example). To do so, we make the use of the Unobservable closure of $\mathcal{G}_\Omega$: $\text{UC}(\mathcal{G}_\Omega)$ (Definition 1.3) that preserves the information about $\mathcal{L}_{Q_F}(\Omega)$ membership while abstracting away unobservable events. Next, a self-product $\text{UC}(\mathcal{G}_\Omega) \times \text{UC}(\mathcal{G}_\Omega)$ allows to extract from a trace μ pairs of trajectories of $\mathcal{G}_\Omega$ and to check their $\mathcal{L}_{Q_F}(\Omega)$ membership agreement.

By definition, for all $\mu \in Traces_{\Sigma_o}(\mathcal{G})$, $q_o \xrightarrow{\mu}_o q'$ in $\text{UC}(\mathcal{G}_\Omega)$ if and only $\exists s \in [\![\mu]\!]_{\Sigma_o}$ s.t. $q \xrightarrow{s} q'$ in M. Consider now $\text{UC}(\mathcal{G}_\Omega) = (Q', \Sigma_o, \rightarrow_o, q_o)$ and let $\Gamma = \text{UC}(\mathcal{G}_\Omega) \times \text{UC}(\mathcal{G}_\Omega)$ be the LTS $(Q' \times Q', \Sigma_o, \rightarrow_\Gamma, (q_o, q_o)$. By definition of UC and synchronous product, if $\mu \in Traces_{\Sigma_o}(\mathcal{G})$ and $(q_o, q_o) \xrightarrow{\mu}_\Gamma (q, q')$ there exists $s, s' \in [\![\mu]\!]_{\Sigma_o}$ s.t. $q_o \xrightarrow{s} q$ and $q_o \xrightarrow{s'} q'$ in $\mathcal{G}_\Omega$.



Definition 2.3 *Given Γ defined as above.*

- *We say that $(q, q') \in Q' \times Q'$ is* Ω-determined *whenever $q \in Q \times Q_F \iff q' \in Q \times Q_F$. Otherwise, it is called* undetermined.
- *A path in Γ is called an* n-undetermined path *if it contains $n+1$ consecutive Ω-undetermined states (thus n events between them).*
- *A path in Γ is an* undetermined cycle *if it is a cycle which states are all undetermined.*

We now show the relation between $\Omega(n)$-diagnosability and the existence of n-undetermined paths.

Theorem 2.2 *$\mathcal{G}$ is Ω-diagnosable if and only if there exists n such that Γ contains no reachable n-undetermined path.*

Observe also that the presence of an undetermined cycle in the twin machine Γ induces the presence of an ambiguous cycle in the diagnoser $Det_{\Sigma_o}(\mathcal{G}_\Omega)$. However, the converse is not true, therefore, the presence of cycles on ambiguous sates in the diagnoser only provides a

sufficient condition for diagnosability. This property of diagnosers was first considered as a necessary and sufficient condition in [SSL+95], and then corrected in [SSL+96].

Based on theorem 2.2 and on the fact that Γ is finite state, we conclude that:

Corollary 2.1 *$\mathcal{G}$ is not Ω-diagnosable if and only if Γ contains a reachable undetermined cycle.*

Using Proposition 2.1 and the construction of Γ, verifying diagnosability amounts to check the existence of reachable undetermined cycles in Γ, retrieving the idea of the algorithm of [YL02, JHCK01]. By Corollary 2.1 and Theorem 2.2, we obtain:

Corollary 2.2 *If $\mathcal{G}$ is Ω-diagnosable, then $\mathcal{G}$ is $\Omega(n+1)$-diagnosable, and not $\Omega(n)$-diagnosable where n is the length of the longest undetermined path of Γ.*

We now summarize the procedure to determine whether $\mathcal{G}$ is Ω-diagnosable: we perform a depth first search on Γ which either exhibits an undetermined cycle or ends by having computed the length of the longest undetermined sequence. Obviously, this procedure has linear cost in the size of Γ.

Example 2.2 *In order to illustrate the construction of Γ, let us come back to the Example 2.1. $\text{UC}(\mathcal{G}_\Omega)$ is given in Figure 2.8 (left-hand side). The rectangles correspond to the marked states. Now $\Gamma = \text{UC}(\mathcal{G}_\Omega) \times \text{UC}(\mathcal{G}_\Omega)$ is given in Figure 2.8 (right-hand side).*

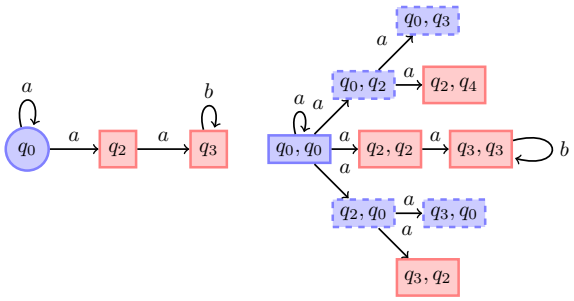


Figure 2.8: $\text{UC}(\mathcal{G}_\Omega)$ and Γ for the LTS of Figure 2.7

The tuples $\{(q_0, q_2), (q_2, q_0), (q_0, q_3), (q_3, q_0)\}$ in Γ are undetermined. Now it is easy to show that there is no undetermined cycle, which according to Corollary 2.1 ensures that $\mathcal{G}$ is Ω_f-diagnosable. Indeed, as soon as f is triggered, b is observed after the occurrence of a finite number of observable events (bounded by 3). Thus the observation of b surely indicates that f occurred in the past.

A contrario, consider the LTS $\mathcal{G}'_\Omega$ in Figure 2.9. Γ' given in Figure 2.10 has undetermined cycles (in $(1,3)$ and $(3,1)$), thus, $\mathcal{G}'_\Omega$ is not Ω_f-diagnosable. In fact, for any n, the trajectories $s_1 = a^n$ and $s_2 = f.a^n$ are both compatible with $\mu = a^n$, while $s_1 \notin \mathcal{L}_F(\Omega_f)$, whereas $s_2 \in \mathcal{L}_F(\Omega_f)$.

Incidentally as previously mentioned, this example proves that Ω-diagnosability cannot be checked directly on the diagnoser. In fact the diagnosers for $\mathcal{G}'$ and Ω_f (Figure 2.9, right) and for $\mathcal{G}$ for Ω (Figure 2.7, right) are isomorphic, and $\mathcal{G}$ is Ω_f-diagnosable while $\mathcal{G}'$ is not.

◇

Figure 2.9: $\mathcal{G}'$ and its associated diagnoser w.r.t. Ω_f

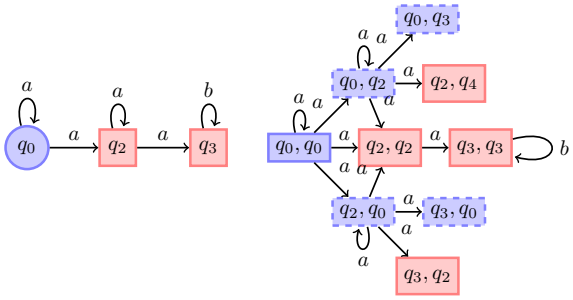


Figure 2.10: Γ' for the LTS $\mathcal{G}'$ of Example 2.1

2.1.4 conclusion

A supervision pattern is an automaton, like the ones used in many different domains (verification, model-based testing, pattern matching, etc), in order to unambiguously denote a formal language. As illustrated in this section, the fault-occurrence diagnosis is a particular case of pattern diagnosis, but patterns are also useful to describe more general objectives, as shown in subsections 2.1.1. The concept of supervision patterns is even more attractive in the sense that patterns can be composed using usual combinators inherited from language theory (union, intersection, concatenation, etc.). Adopting the behavioral properties point of view on the patterns leads to the attempt to diagnose any linear time pure past temporal formulas [LS95]. It is clear that the properties we consider do not meet the LTL definable properties handled by [JK04]. We now turn to technical aspects of the approach. We have insisted on what the semantics of a trace is: a trace denotes the set of trajectories which project onto this trace and that necessarily end up with an observable event. Consequences of this choice are manifold in the definitions of Ω-diagnosability, $Det_{\Sigma_o}(\mathcal{G}_\Omega)$ and $\textsc{Uc}(\mathcal{G})$. We could have chosen another semantics, impacting on the related definitions accordingly: for example we could have considered the set of trajectories which project onto this trace. What is mostly important is the accurate match between the semantics for traces and the other definitions: hence we avoid displeasing discrepancies to determine precisely the Diagnosability Bound, and even better, we have a clear proof for the correctness of the synthesis algorithm. However, we believe our choice is the most natural when admitting that the diagnosis function implemented online as an output verdict is reactive to an observable move of the system.

2.2 Predictability of Stable Supervision patterns

Compared with Section 2.1, this section takes the diagnosis problem one step ahead, and considers the problem of predicting (faulty) sequence patterns in the behavior of a partially-observed discrete-event system (DES). If it is possible to predict the occurrences of a pattern that describes event sequences that lead or cause the system to fail or malfunction, then the system operator can be warned in time to halt the system, take preventative measures, or otherwise to reconfigure the system.

The problem of prediction of patterns in DES as considered here builds on and extends two prior studies: (i) the problem of predictability of single event occurrences in partially-observed DES studied in [GL06b] and (ii) the problem of diagnosis of sequence patterns in DES studied in [C22] (presented in Section 2.1) and [GL06a]. The consideration of patterns in the context of the property of predictability requires more than technical modifications to the two works mentioned above; this is true especially for the development of a polynomial-time test for pattern predictability, as will be seen later in this section. The notion of predictability we here consider in this chapter is different from prior works on other notions of predictability in [Cao89], [SRBT91], [FH99]. Other references on predictability in DES or discrete event simulation systems are [BM95], [CR90], [Dec98], [MM06], [QHF02]. The results presented in this chapter are related to [JK04] in which the authors considered the notion of inevitability. Indeed, in this paper, the authors are interested in diagnosing inevitability, thus in detecting that a pattern will be inevitably satisfied within a bounded number of observations after this inevitability. In contrast, in the context of this chapter, predictability means detecting inevitability strictly before its occurrence.

In this section, we introduce the notion of predictability of a pattern. $\mathcal{G}$ and Ω are as defined in the preceding section. Roughly speaking, a pattern is predictable if it is always possible to detect the future recognition of the pattern, strictly before this recognition, only based on the observations. We explain the idea of predictability in the following example.

Example 2.3 *Consider the LTS $\mathcal{G}_1$ shown in Fig.2.11. The set of events is $\Sigma = \{f_1, f_2, a, b, c\}$. Let $\Sigma_{uo} = \{a, f_1, f_2\}$ be the set of unobservable events. We assume that the pattern under consideration is either the occurrence of f_1 followed by f_2 or the one of f_2 followed by f_1. This pattern is given in Fig. 2.2 and denoted by Ω in this example. By a simple inspection of $\mathcal{G}_1$ in Fig.2.11, we see that there are three branches in which f_1*

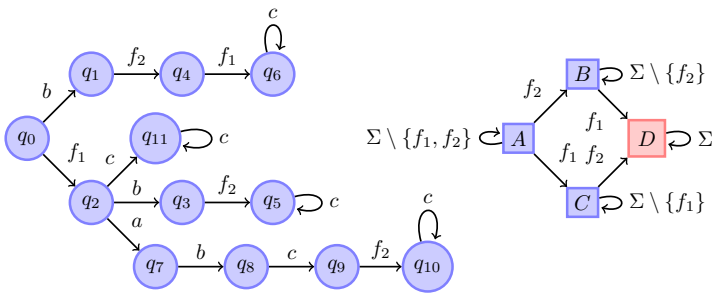


Figure 2.11: LST $\mathcal{G}_1$

is followed by f_2 (or f_2 followed by f_1): $b.f_2.f_1.c^*$, $f_1.b.f_2.c^*$ and $f_1.a.b.c.f_2.c^*$. The other branch, which does not satisfy the pattern, records $f_1.c^*$. Thus, after the observation of $b.c.c$, we know for sure that the pattern has been recognized. $\mathcal{G}_1$ is thus Ω-diagnosable. Moreover, we can also predict the recognition of the pattern ahead of time. Indeed, if we do observe b then we know for sure that the pattern has not occurred so far but will occur in the future. After b, the system is either in state 1, 8 or 3. After the occurrence of c, then $\mathcal{G}_1$ is either in 5, 6 or 9. In the first two cases, the pattern has occurred. In the third case, we need to wait for a second c to occur to be sure that it occurred (the system is then either in state 5, 6, 10, in which the pattern has surely occurred). ◇

2.2.1 Definition of predictability

We now formally define the predictability of patterns. As previously said, this generalizes the predictability definition introduced in [GL06b] by considering sequence patterns instead of single events.

Definition 2.4 *An LTS $\mathcal{G}$ is* $\Omega(n)$-predictable, *where $n \in \mathbb{N}$, whenever*

$$\exists n \in \mathbb{N}, \forall s \in \mathcal{L}(\mathcal{G}) \cap \mathcal{L}_{Q_F}(\Omega).\Sigma^*.\Sigma_o, \exists t \in [\mathcal{L}(\mathcal{G}) \cap \Sigma^*.\Sigma_o] \cup \{\epsilon\},\ t < s\ \wedge\ t \notin \mathcal{L}_{Q_F}(\Omega)$$

such that

$$\forall u \in [\![P_{\Sigma_o}(t)]\!]_{\Sigma_o}, \forall v \in \mathcal{L}(\mathcal{G})/u, |P_{\Sigma_o}(v)| \geq n \implies u.v \in \mathcal{L}_{Q_F}(\Omega)$$

$\mathcal{G}$ is said to be Ω-predictable *if it is* $\Omega(n)$-predictable *for some $n \in \mathbb{N}$.*

The definition means that for any trajectory s recognized by the pattern, there exists a strict prefix t not recognized by the pattern, such that any trajectory u compatible with observation $P_{\Sigma_o}(t)$ will inevitably be extended into a trajectory $u.v$ recognized by the pattern. That is, upon observation of $P_{\Sigma_o}(t)$, one can already predict that the pattern, while not yet recognized, will inevitably be recognized; see (see Figure 2.12) for a conceptual representation.

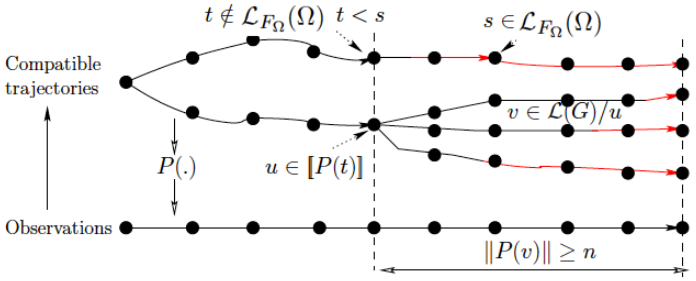


Figure 2.12: Intuitive explanation of the predictability

Let us now relate the notion of Ω-predictability to the notion of Ω-diagnosability of Section 2.1.

Proposition 2.4 *Given a system G and a supervision pattern Ω, if G is Ω-predictable, then G is Ω-Diagnosable.*

The proof of the proposition immediately follows from the respective definitions of these properties. (Further details on pattern diagnosability can be found in [C22], [GL06a]). Note that the converse of this proposition is false as shown by the following example:

Example 2.4 *We consider a variation of Example 2.3. The new system is given by the LTS $\mathcal{G}_2$ represented in Fig. 2.11 (and reproduced here in the right-hand side of Fig. 2.11 for completeness. The pattern and the partition of the event sets into observable and unobservable events are the same as in Example 2.3. The occurrence of the pattern Ω is predictable in $\mathcal{G}_2$.*

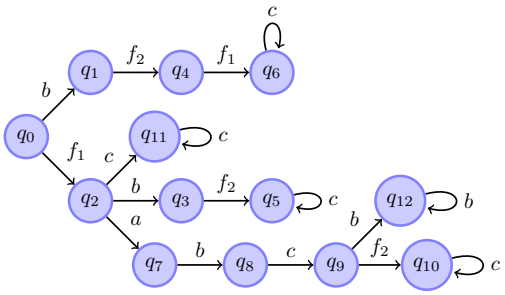


Figure 2.13: LST $\mathcal{G}_2$

It is easy to show that $\mathcal{G}_2$ is Ω-diagnosable. However, $\mathcal{G}_2$ is not Ω-predictable. Indeed, because of the sequence $f_1.a.b.c.b^*$ that is not recognized by the pattern, one can not be sure that the pattern will occur by observing the trace b. Further, if we observe the trace $b.c$, it may be too late, as the system may have trigger the sequence $f_1.b.f_2.c$, which is a sequence recognized by the pattern. ◇

2.2.2 Verification of Predictability

In this section, we present an off-line algorithm to verify the property of predictability. In [GL06a], where the problem of predictability of single event occurrences is considered, the algorithm for the verification of predictability is based on the construction of a diagnoser automaton. In this part, we propose to adapt the polynomial-time verification of diagnosability based on verifier LTS [JHCK01, YL02] to the verification of predictability. In Section 2.1.3, we provided a technique based on verifiers to verify the diagnosability of a pattern in polynomial time in the number of states of the system. In the case of diagnosability, the verifier identifies the existence, if any, of strings that violate the definition of this property. We adopt this underlying principle in building the algorithm for verification of predictability of patterns. By definition of predictability, the trajectories that violate predictability are the ones that: (i) are accepted by the pattern, (ii) end with observable events, and (iii) have the property that none of their prefixes are sufficient to predict the occurrence of the pattern. That is, for all the prefixes of such trajectories, there exists a trajectory that produces the very same observation as the prefix but whose continuation does not contain the pattern. Thus, if we find one or more trajectories violating predictability as was just described, then we can conclude that at least one occurrence of the pattern is not predictable

in the system. It is possible to transform the search for these trajectories into a search for states in an LTS that carry information on acceptance, inevitability, and projections, for trajectories leading to them. This avoids the construction of the diagnoser, providing an algorithm with polynomial complexity.

In the remainder of this section, we decompose the test of predictability into 5 steps, each of them corresponding to particular operations requested to perform the test. Assume that the predictability problem is given by $\mathcal{G} = (Q, \Sigma, \rightarrow, q_0)$ and $\Omega = (Q_\Omega, \Sigma, \rightarrow_\Omega, q^0_\Omega)$ equipped with F_Ω.

Step 1. It consists in constructing the synchronous product $\mathcal{G}_\Omega = \mathcal{G} \times \Omega = (Q_{\mathcal{G}_\Omega}, \Sigma, \rightarrow_{\mathcal{G}_\Omega}, q^0_{\mathcal{G}_\Omega})$ and the final state set $F = Q \times F_\Omega$. By the properties of the synchronous product, and as Ω is complete (thus $\mathcal{L}(\Omega) = \Sigma^*$) we get $\mathcal{L}(\mathcal{G}_\Omega) = \mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\Omega) = \mathcal{L}(\mathcal{G})$ and $\mathcal{L}_F(\mathcal{G}_\Omega) = \mathcal{L}(\mathcal{G}) \cap \mathcal{L}_{F_\Omega}(\Omega)$ meaning that the accepted trajectories of $\mathcal{G}_\Omega$ in F are exactly the trajectories of $\mathcal{G}$ accepted by Ω in F_Ω. Finally note that F is stable in $\mathcal{G}_\Omega$ as both Q and F_Ω are stable by assumption.

Step 2. It consists in computing $Inev_{\mathcal{G}_\Omega}(F)$ (see Eq. 1.9) and to consider the following partition of the state space $Q_{\mathcal{G}_\Omega}$: $Q_{\mathcal{G}_\Omega} = N \cup I \cup F$ where $I = Inev_{\mathcal{G}_\Omega}(F) \setminus F$ is the set of states not belonging to F but from which F is unavoidable, and $N = Q_{\mathcal{G}_\Omega} \setminus Inev_{\mathcal{G}_\Omega}(F_{\mathcal{G}_\Omega})$ is the set of states from which F is avoidable.

The diagram of Fig. 2.14 shows how $\mathcal{G}_\Omega$ evolves from one part to another according to the set of states it belongs to. This diagram illustrates the fact that even though the system can remain forever in N-states, if it reaches an I-state, then after a finite number of steps[2] the system will eventually evolve into an F-state.

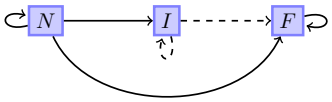


Figure 2.14: Moves in the partition of $Q_{\mathcal{G}_\Omega}$

Example 2.5 *The construction of the LTS $\mathcal{G}_{2_\Omega}$ of Fig. 2.13, is illustrated by Fig. 2.15, with N-states in blue, F-states in red, whereas the I-states are those with dashed-line borders.* ◇

Step 3. Construct the Unobservable-Closure (see Def. 1.3) $\text{UC}(\mathcal{G}_\Omega) = (Q_{\mathcal{G}_\Omega}, \Sigma_o, \rightarrow_{\text{UC}}, q^0_{\mathcal{G}_\Omega})$ of $\mathcal{G}_\Omega$ with final state set F and preservation of the partition $Q_{\mathcal{G}_\Omega} = N \cup I \cup F$. Using this partition, we also label states in $(Pre^{\exists}_{\text{UC}(\mathcal{G}_\Omega)}(F) \setminus F)$, corresponding to the set of states having an immediate successor in F, but not belonging to F. It thus characterizes the maximal prefixes t of trajectories of $\mathcal{G}$ not recognized by Ω in F_Ω, *i.e.* their observation is the last chance to predict acceptance. By the properties of UC, we get $\mathcal{L}(\text{UC}(\mathcal{G}_\Omega)) = P_{\Sigma_o}(\mathcal{L}(\mathcal{G}_\Omega)$ and $\mathcal{L}_F(\text{UC}(\mathcal{G}_\Omega) = P_{\Sigma_o}(\mathcal{L}_F(\mathcal{G}_\Omega))$ and $\mathcal{L}_F(\text{UC}(\mathcal{G}_\Omega) = P_{\Sigma_o}(\mathcal{L}_F(\mathcal{G}_\Omega))$ which

[2] represented by dashed lines

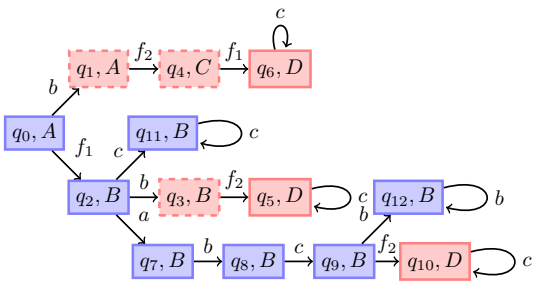


Figure 2.15: The LTS $\mathcal{G}_{1\Omega}$

in some sense means that Uc preserves information on (the inevitability of the) recognition by the pattern, while abstracting away internal events. To illustrate this point, we use Example 2.4. The Unobservable-Closure of $\mathcal{G}_{2\Omega}$ is described in Fig. 2.16 where states in $(Pre^{\exists}_{\mathrm{UC}(\mathcal{G}_\Omega)}(F) \setminus F)$ are those with dashed-line borders.

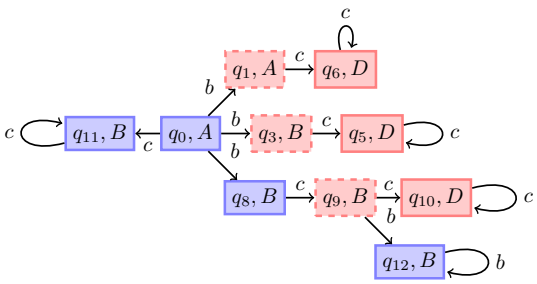


Figure 2.16: The LTS $\mathrm{Uc}(\mathcal{G}_{2_\Omega})$

Step 4. Construct the LTS $\Gamma = \mathrm{Uc}(\mathcal{G}_\Omega) \times \mathrm{Uc}(\mathcal{G}_\Omega) = (Q_{\mathcal{G}_\Omega} \times Q_{\mathcal{G}_\Omega}, \Sigma_o, \rightarrow_\Gamma, (q^0_{\mathcal{G}_\Omega}, q^0_{\mathcal{G}_\Omega}))$. The role of Γ is to synchronize trajectories of $\mathcal{G}_\Omega$ with same trace μ.

Step 5. Check the predictability condition given by the theorem 2.3 below. This completes the statement of the verification algorithm.

Theorem 2.3 *$\mathcal{G}$ is Ω-predictable iff $(Pre^{\exists}_{\mathrm{UC}(\mathcal{G}_\Omega)}(F) \setminus F \times N$ is not reachable from $(q^0_{\mathcal{G}_\Omega}, q^0_{\mathcal{G}_\Omega})$ in Γ.*

A reachable state in $(Pre^{\exists}_{\mathrm{UC}(\mathcal{G}_\Omega)}(F) \setminus F) \times N$ characterizes two trajectories with same observation, one where F_Ω is avoidable, and the other one where it was the last chance to predict acceptance. Notice that symmetry in Γ allows us to simplify the condition, but in practice, also reachable states in $N \times (Pre^{\exists}_{\mathrm{UC}(\mathcal{G}_\Omega)}(F) \cap \setminus F)$ should be detected.

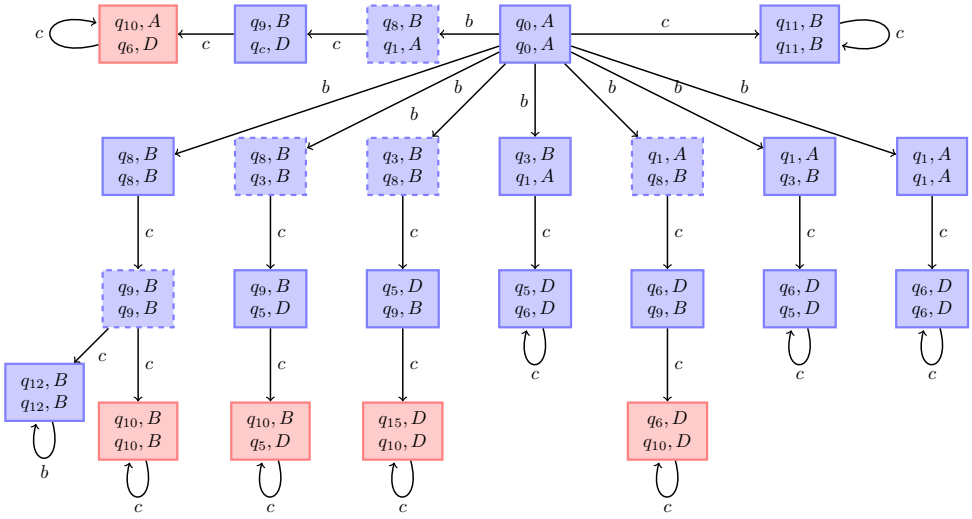


Figure 2.17: The LTS Γ_2 derived from $\mathcal{G}_{2_\Omega}$

Example 2.6 *Figures 2.17 describe the construction of* Γ_2 *for* G_2 *with the pattern* Ω. *The 6 states with dashed-line borders in* Γ_2 *are states in* $(Pre^{\exists}_{\mathrm{UC}(\mathcal{G}_{2_\Omega})}(F) \setminus F) \times N$ *and prove that* G_2 *is not* Ω*-predictable. For example, the string* $b.f_2.f_1.c$ *with observation* $b.c$ *already leads to acceptance, but acceptance is not predictable because of trajectories in* $f_1.a.b.c.b^*$ *Finally, one can note that* G_2 *is* Ω*-diagnosable, as in* Γ_2*, there is no cycle of states of the form* $(I \cup N) \times F$. ◇

To summarize this section we now outline a possible algorithm *Test_Pred* allowing to test the predictability of a system G w.r.t. a pattern Ω:

Algorithm 1: Test_Pred($\mathcal{G}$,Ω)

input : $\mathcal{G}$ and Ω.
output: Is Ω predictive or not w.r.t. $\mathcal{G}$?

```
begin
    G_Ω = G × Ω
    Compute the states set F, I and N on G_Ω
    Compute UC(G_Ω) and the state set (Pre^∃_UC(G_Ω)(F) \ F)
    Γ = UC(G_Ω) × UC(G_Ω)
    return ( (Pre^∃_UC(G_Ω)(F) \ F) × N = ∅ in Γ)
end
```

Proposition 2.5 *Using algorithm Test_Pred*($\mathcal{G}$,Ω)*, the overall complexity of the test of predictability is quadratic in the product of the sizes of* $\mathcal{G}$ *and* Ω.

The first step is the construction of $\mathcal{G}_\Omega = \mathcal{G} \times \Omega$ which is linear in $\|\mathcal{G}\| \times \|\Omega\|$ where $\|\mathcal{G}\|$ and $\|\Omega\|$ are the sizes of $\mathcal{G}$ and Ω in terms of states and transitions. In the second step, one needs to compute $Inev_{\mathcal{G}_\Omega}(F)$, to partition $Q_{\mathcal{G}_\Omega}$ into $N \cup I \cup F$. This is linear in $\|\mathcal{G}_\Omega\|$. The third step consists in constructing $\text{UC}(\mathcal{G}_\Omega)$, and labelling of states by $(Pre^{\exists}_{\text{UC}(\mathcal{G}_\Omega)}(F) \setminus F)$. It is linear in $\|\mathcal{G}_\Omega\|$. The construction of $\Gamma = \text{UC}(\mathcal{G}_\Omega) \times \text{UC}(\mathcal{G}_\Omega)$ in the fourth step is quadratic in the size of $\text{UC}(\mathcal{G}_\Omega)$. Checking that no reachable state is in $(Pre^{\exists}_{\text{UC}(\mathcal{G}_\Omega)}(F) \setminus F) \times N$ (Step 5) can be done during this construction. The overall complexity of the verification of predictability is thus quadratic in the product of the sizes of $\mathcal{G}$ and Ω.

2.2.3 The P_diagnoser

In this section, we now explain how to construct from $\mathcal{G}$ and Ω a special kind of diagnoser, called P_diagnoser (for Prediction Diagnoser) that can be used on-line to predict the recognition of a pattern. We emphasize some properties of this P_diagnoser when the system is predictable w.r.t. a given pattern Ω.

To do so, let us consider $\mathcal{G}_\Omega$ as defined in Section 2.2.2 (Step 1) and build $Det_{\Sigma_o}(\mathcal{G}_\Omega) = (\mathcal{X}, \Sigma_o, \rightarrow_d, X^0)$ as defined in Def. 1.5. Remember that this can be done by reusing $\text{UC}(\mathcal{G}_\Omega)$ and apply a subset construction. We then have $\mathcal{L}(De_{\Sigma_o}t(\mathcal{G}_\Omega)) = P_{\Sigma_o}(\mathcal{L}(\mathcal{G}_\Omega)) = P_{\Sigma_o}(\mathcal{L}(G))$ thus for any observation $\mu \in P_{\Sigma_o}(\mathcal{L}(G))$, $\delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) = \{\delta_{\mathcal{G}_\Omega}(q^0_{\mathcal{G}_\Omega}, [\![\mu]\!]_{\Sigma_o})\}$.

The P_diagnoser for predictability $\texttt{Predict}_\Omega$ is defined from $Det_{\Sigma_o}(\mathcal{G}_\Omega)$ as follows. For any observation μ in $Traces_{\Sigma_o}(\mathcal{G})$, let $\texttt{Predict}_\Omega(\mu)$ be:

$$\begin{cases} Yes & \text{if} \quad \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \subseteq F \\ Pred & \text{if} \quad \delta_{Det_{\Sigma_o}t(\mathcal{G}_\Omega)}(X^0, \mu) \subseteq I \\ Pred\_F & \text{if} \quad \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \subseteq (I \cup F) \ \wedge \ \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \cap I \neq \emptyset \\ & \quad \wedge \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \cap F \neq \emptyset \\ No & \text{if} \quad \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \subseteq N \\ Pred\_N & \text{if} \quad \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \subseteq (I \cup N) \ \wedge \ \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \cap I \neq \emptyset \\ & \quad \wedge \delta_{Det_{\Sigma_o}(\mathcal{G}_\Omega)}(X^0, \mu) \cap N \neq \emptyset \\ ? & \text{otherwise.} \end{cases} \tag{2.6}$$

To clarify the verdicts given by the P_diagnoser, we now adopt a language point of view. Using the partition of $Q_{\mathcal{G}_\Omega} = N \cup I \cup F$, we can partition $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_\Omega)$ into three different languages

$$\mathcal{L}(\mathcal{G}) = \mathcal{L}_N \cup \mathcal{L}_I(\mathcal{G}_\Omega) \cup \mathcal{L}_F(\mathcal{G}_\Omega)$$

where $\mathcal{L}_N = \mathcal{L}(\mathcal{G}) \setminus (\mathcal{L}_F(\mathcal{G}_\Omega) \cup \mathcal{L}_I(\mathcal{G}_\Omega))$. Based on this partition, we then have the following results that directly derive from the definition on the P_diagnoser

- $\texttt{Predict}_\Omega(\mu) = Yes$ if $[\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_F(\mathcal{G}_\Omega)$

- $\texttt{Predict}_\Omega(\mu) = Pred$ if $[\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_I(\mathcal{G}_\Omega)$
- $\texttt{Predict}_\Omega(\mu) = Pred\_F$ if
  - $[\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_I(\mathcal{G}_\Omega) \cup \mathcal{L}_F(\mathcal{G}_\Omega)$, and
  - $[\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_I(\mathcal{G}_\Omega) \neq \emptyset$, and
  - $[\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_F(\mathcal{G}_\Omega) \neq \emptyset$
- $\texttt{Predict}_\Omega(\mu) = No$ if $[\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_N$
- $\texttt{Predict}_\Omega(\mu) = Pred\_N$ if
  - $[\![\mu]\!]_{\Sigma_o} \subseteq \mathcal{L}_N \cup \mathcal{L}_I(\mathcal{G}_\Omega)$, and
  - $[\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_N \neq \emptyset$, and
  - $[\![\mu]\!]_{\Sigma_o} \cap \mathcal{L}_I(\mathcal{G}_\Omega) \neq \emptyset$

Yes means that all the trajectories in $[\![\mu]\!]_{\Sigma_o}$ lie in $\mathcal{L}_{F_\Omega}(\Omega)$, which means that the observation μ only corresponds to trajectories recognized by the pattern. $Pred$ means that trajectories compatible with μ are not recognized by the pattern so far, but all their extensions will inevitably be after a bounded number of observations. No simply means that all trajectories compatible with μ can still be extended without being recognized by the pattern. $Pred\_F$ means that the recognition of the pattern is unavoidable but some trajectories compatible with the observation are already recognized by the pattern. $Pred\_N$ means that the observed trace corresponds to some trajectories where recognition of the pattern is unavoidable, but others where this is false.

Example 2.7 *Back to Examples 2.3 and 2.4, the P_diagnosers (as well as their verdicts) obtained by determinization of $\mathcal{G}_{1\Omega}$ and $\mathcal{G}_{2\Omega}$ respectively for $\mathcal{G}_1$ and $\mathcal{G}_2$ with the pattern Ω given in Figure 2.18.* ◇

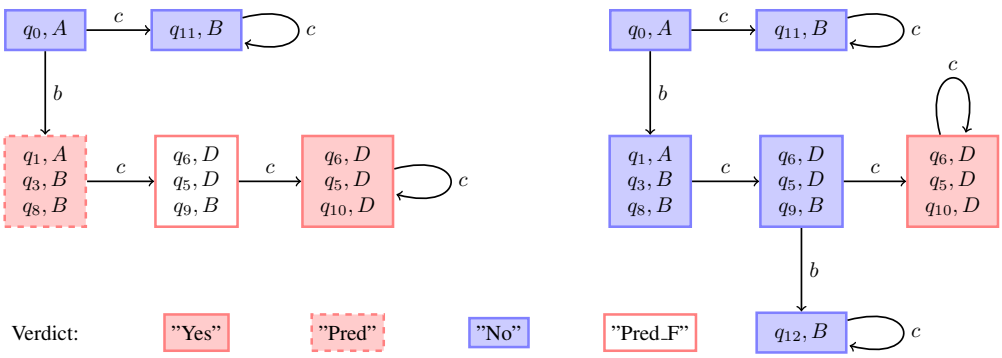


Figure 2.18: The P_diagnosers of $\mathcal{G}_1$ (left-hand side) and $\mathcal{G}_2$ (right-hand side)

We have the following results about predictors.

Proposition 2.6 *If $\mathcal{G}$ is $\Omega(n)$-predictable, then*

(i) $\texttt{Predict}_\Omega(\mu) = Pred \Rightarrow (\forall \mu' \in P_{\Sigma_o}(\mathcal{L}(\mathcal{G}))/\mu, \|\mu'\| \geq n \Rightarrow \texttt{Predict}_\Omega(\mu\mu') = Yes)$

(ii) $\texttt{Predict}_\Omega(\mu) =$ *"YES"* $\Rightarrow \exists \mu' < \mu$ *s.t.* $\texttt{Predict}_\Omega(\mu') = Pred$ *and* $\|\mu\| - \|\mu'\| \leq n$

(iii) $\forall \mu \in P_{\Sigma_o}(\mathcal{L}(\mathcal{G}))$, $\texttt{Predict}_\Omega(\mu) \neq ?$ $\diamond$

Point (i) simply says that whenever the P_diagnoser emits the verdict $Pred$, then after at most n observations, the P_diagnoser will emit the verdict Yes. Point (ii) emphasizes the fact that if the P_diagnoser emits the verdict Yes, then the P_diagnoser emitted, in the past, the verdict $Pred$. Point (iii) shows that the verdict ? can not be emitted as far as the system is Ω-predictable. Indeed, if this verdict is emitted, it would say that there is a trajectory compatible with μ that is recognized by the pattern without having being detected, which contradicts the fact that the system is predictable.

Figure 2.19 explains the possible evolutions of the verdicts emitted by the P_Diagnoser whenever the system is predictable.

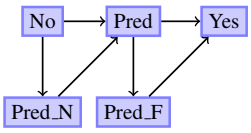


Figure 2.19: Possible evolutions of the P_diagnoser verdicts.

Comparison with [JK04]. The results we presented in this paper are very close to the work of Jiang and Kumar [JK04] who introduced the notion of *inevitability*. Indeed, in [JK04], the authors are interested in diagnosing inevitability, thus in detecting that a pattern will be inevitably satisfied, within a bounded number of observations *after* after this inevitability[3], while prediction means detecting inevitability strictly before satisfaction. It should also be noted that, despite the fact that the diagnoser for inevitability may give earlier verdicts than the diagnoser for satisfaction, the diagnosability of inevitability is strictly equivalent to diagnosability of satisfaction. In contrast, we have shown that predictability implies diagnosability, but not the converse. One consequence of their definition of diagnosability is that the on-line diagnoser is not able to infer whether the pattern has been actually recognized or not. Indeed, compared to the P_diagnoser, the verdicts $Pred$, $Pred\_F$ and F are merged into a single verdict.

2.2.4 Conclusion

In this section, we have presented a generalized notion of predictability of patterns in discrete event systems. Predictability of patterns defines the ability to detect the recognition

[3]This work considers properties specified in Linear Temporal Logic (LTL) which are more expressive as patterns. However, their notion of pre-diagnosability required for diagnosability means that the property reduces to a stable property on the system (can be checked on a finite trajectory).

of this pattern strictly before its actual occurrence. Based on rather standard operations on LTSs, we provide a test of polynomial complexity to verify the predictability property as well as an algorithm allowing to construct a diagnoser for the purpose of the on-line prediction of the pattern.

2.3 Diagnosis of Intermittent Failures

So far, we considered the diagnosis of stable supervision patterns (i.e. the faulty behaviour is absorbing: once the system becomes faulty for a particular execution, it remains faulty for all possible extensions of this execution. However, in many systems, faulty behaviour might occur intermittently, with faulty sequences that can be extended to sequences after which the system is repaired. It is worthwhile noticing that the procedure described in the first section of this Chapter, can not be used as such to diagnose the occurrence of such intermittent (or repairable) behaviors.

2.3.1 Notations and preliminary results

In this section, we shall assume that the system to be diagnosed is given by a deterministic LTS $\mathcal{G} = (Q, q_0, \Sigma, \longrightarrow)$ for which the set of states is partitioned into two subsets $Q_N \uplus Q_F$, and let us name Q_N normal (or safe) states and Q_F faulty states, to help intuition. As usual, not all, but only a part of the events is observable: $\Sigma = \Sigma_o \uplus \Sigma_u$. The aim of the diagnosis is then to be able to detect that the system is in a faulty state after a given observation.

The *faulty language* of $\mathcal{G}$ is derived from *faulty sequences*, *i.e.* sequences that terminate in a faulty state: $\mathcal{L}_F(\mathcal{G}) = \{s \in \Sigma^*,\ \delta_{\mathcal{G}}(q_0, s) \in Q_F\}$. The *normal (safe) language* of $\mathcal{G}$ is defined similarly, and denoted $\mathcal{L}_N(\mathcal{G})$. As $\mathcal{G}$ is deterministic $\mathcal{L}_N(\mathcal{G}) \cap \mathcal{L}_F(\mathcal{G}) = \emptyset$, or $\mathcal{L}_N(\mathcal{G}) \uplus \mathcal{L}_F(\mathcal{G}) = \mathcal{L}(\mathcal{G})$.

Remark 2.1 *One can note that the faulty language can be obtained by considering a supervision pattern as described in the first section without the asumption of stability for the final states and by performing the product between the system and this supervision pattern.*

When faults are non permanent in $\mathcal{G}$, that is when there exist transitions from Q_F to Q_N, one may nevertheless be interested in detecting that some transient fault (i.e., fault that has occurred but was then repaired) has occurred. This can again be captured by an obvious transform of $\mathcal{G}$ into $\mathcal{G}'$, that adds memory to states of $\mathcal{G}$ to propagate the fact that a fault occurred sometimes in the past. With the assumption that $\mathcal{G}$ is deterministic, this amounts to saturating the fault language of $\mathcal{G}$: $\mathcal{L}_F(\mathcal{G}') = \mathcal{L}_F(\mathcal{G})\,\Sigma^* \cap \mathcal{L}(\mathcal{G})$. This idea is a variant of the pattern recognition introduced in section 2.1. It was used in [JKG03] to track the occurrence of k transient faults. It is also present in [CLT04] under the names of O-diagnosis (detection of the occurrence of a fault) and I-diagnosis (detection of the occurrence of a repair). All these notions are thus variants of the classical diagnosis approach, even if they are recast in the context of transient failures. In [CLT04], the authors propose a "memory LTS" that can be composed with a specification to remember occurrences of faults and repairs. However, even if fault repair is considered, the LTS propagates the information that a fault occurred. In the next section, we consider a different setting, where diagnosis is considered as accurate if it detects a fault before it is repaired.

2.3.2 Diagnosis and T-diagnosability

We still consider a Σ_o-live deterministic LTS $\mathcal{G}$, and assume that some faults in $\mathcal{G}$ can be repaired, i.e. $\mathcal{G}$ contains transitions from Q_F to Q_N, or equivalently that the fault language $\mathcal{L}_F(\mathcal{G})$ is not extension-closed. The diagnoser of an observed sequence $\mu = P_{\Sigma_o}(s)$ produced by some sequences s of $\mathcal{G}$ is defined as in Eq. (2.5). However, we reinforce the diagnosability criterion for $\mathcal{G}$ by requiring that, when some fault occurs, it is still detected in finite time, but also *before it is repaired*.

Let us first introduce some notation. We denote by

$$\mathcal{L}_F^{min}(\mathcal{G}) = \{s.\sigma \in \mathcal{L}_F(\mathcal{G}) \mid s \notin \mathcal{L}_F(\mathcal{G}) \wedge \sigma \in \Sigma\}$$

the set of minimal faulty words of $\mathcal{G}$, i.e., words that correspond to a run ending with a transition from a normal state to a faulty one in $\mathcal{G}$. For a word $s \in \mathcal{L}_F(\mathcal{G})$, let $s \rightarrow s.t \in \mathcal{L}_F(\mathcal{G})$ denote the continuous presence of a fault along t. Formally, $s \rightarrow s.t \in \mathcal{L}_F(\mathcal{G})$ iff $\forall\, t' \leq t,\ s.t' \in \mathcal{L}_F(\mathcal{G})$, where $\leq$ denotes the prefix relation on words.

Definition 2.5 *An LTS $\mathcal{G}$ is* timely diagnosable *(*T-diagnosable *for short) whenever*

$$\begin{array}{c} \forall\, s \in \mathcal{L}_F^{min}(\mathcal{G}),\ \exists\, n \in \mathbb{N},\ \forall\, s.t \in \mathcal{L}(\mathcal{G}), \\ [\ |P_{\Sigma_o}(t)| \geq n \ \ \Rightarrow \ \ \exists\, t' \leq t\ :\ s \rightarrow s.t' \in \mathcal{L}_F(\mathcal{G}) \\ \wedge\ \ [\![s.t']\!]_{\Sigma_o} \subseteq \mathcal{L}_F(\mathcal{G})\ ] \end{array} \tag{2.7}$$

T-diagnosability differs from Def. (2.2) mainly by requiring that the fault that appears in s remains for the whole execution of a prefix t' of t. This notion is illustrated in Fig. 2.20, that depicts several observationally equivalent runs, and shows observation times at which a correct diagnosis/detection can be produced (before repair). Observe that if faults are not repairable, $s \in \mathcal{L}_F^{min}(\mathcal{G})$ implies that $s \rightarrow s.t' \in \mathcal{L}_F(\mathcal{G})$ for every t', and Def. (2.8) reduces to Def. (2.2) (condition $\forall s \in \mathcal{L}_F(\mathcal{G})$ in Def. (2.2) can equivalently be replaced by $\forall s \in \mathcal{L}_F^{min}(\mathcal{G})$). So, in a setting of permanent faults, T-diagnosability is equivalent to diagnosability.

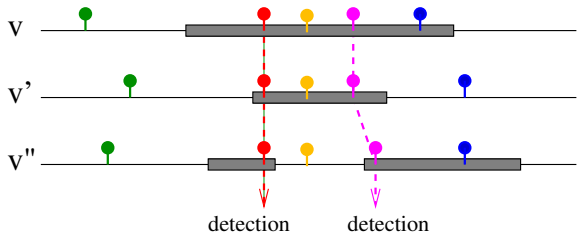


Figure 2.20: A faulty word v and two equivalent words v', v''. The observed labels are represented as pins, and the faulty zones as grey rectangles. Detections correspond to times (in number of observations) where all equivalent words are faulty.

Example 2.8 *Fig. 2.21 illustrates the notion of T-diagnosability. Safe (resp. faulty) states are represented as black (resp. red) boxes. One has $\Sigma = \{a, b, c, d\}$ and $\Sigma_o = \{a\}$. Ignoring the grayed transitions at the bottom, the LTS is T-diagnosable as after the observation*

of sequence a a fault occurred in both runs at the top, and this fault is each time detected before it is repaired since $Diag(a) = F$. By adding the bottom part, T-diagnosability is lost : once a has been observed, one knows for sure that a fault occurred, but no detection can take place before repair, in all runs, as now $Diag(a) =$ "?". ◇

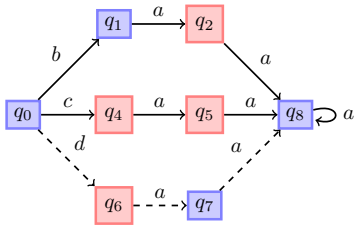


Figure 2.21: A T-diagnosable system, when the path at the bottom is ignored.

2.3.3 Vanishing faults and repairs

T-diagnosability seems to be a reasonable first step towards the ability to count fault occurrences. Unfortunately, this is not the case as it is already apparent in Fig. 2.20 : an LTS with such equivalent sequences can be T-diagnosable, and nevertheless the same observed sequence matches a sequence with one fault (top) and one with two faults (bottom). The situation is even worse. Let us call a *vanishing fault* a fault that occurs and is repaired in the silent part of a sequence of $\mathcal{G}$ (i.e., between two observations), and similarly for a *vanishing repair*. Then, the LTS $\mathcal{G}$ can exhibit runs with an arbitrary number of vanishing faults and repairs without losing its T-diagnosability.
This is illustrated by the example in Fig. 2.22: $\Sigma = \{a, b\}$, $\Sigma_o = \{a\}$. In this LTS $\mathcal{G}$, one has $\texttt{Diag}_\Omega(a) =$ "F". A vanishing repair appears at the end of sequence ab and a vanishing fault at the end of $a.b.b$. Nevertheless, T-diagnosability holds : for $s = a \in L_F^{min}(\mathcal{G})$ one gets immediate fault detection ($t = \epsilon$ works), for $s = a.b^2 \in L_F^{min}(\mathcal{G})$ one has $[\![s]\!]_{\Sigma_o} = \{a\} \subseteq L_F(\mathcal{G})$ so again the fault detection is "immediate" with $t = \epsilon$, and similarly for $s = a.b^4 \in L_F^{min}(\mathcal{G})$.

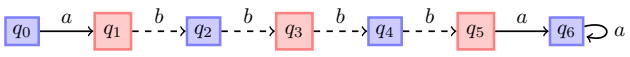


Figure 2.22: An arbitrary number of vanishing faults and repairs may exist in a T-diagnosable automaton.

It is quite counter-intuitive that the "immediate" detection of the fault occurring at $s = a.b^2$ actually relies on the detection of the fault that took place previously, at $s = a$. This phenomenon is due to the fact that T-diagnosability, just as diagnosability, only refers to runs that stop at a visible transition. Everything that happens between observations is almost ignored. For permanent faults, this is harmless: it only shifts the detection by one observation. For repairable faults, it introduces odd phenomena. A natural way to make

fault detection causal (and to open the way to a counting of faults) is thus to forbid the existence of vanishing repairs

$$\nexists\, s = s_1.s_2.a \in \mathcal{L}(\mathcal{G}) \,:\, s_1 \in \mathcal{L}_F(\mathcal{G}) \;\wedge\; s_1.s_2 \in \mathcal{L}_N(\mathcal{G}) \\ \wedge\; a \in \Sigma \;\wedge\; s_1.s_2.a \in \mathcal{L}_F(\mathcal{G}) \;\wedge\; P_{\Sigma_o}(s_2) = \epsilon \tag{2.8}$$

and of vanishing faults

$$\nexists\, s = s_1.s_2.a \in \mathcal{L}(\mathcal{G}) \,:\, s_1 \in \mathcal{L}_N(\mathcal{G}) \;\wedge\; s_1.s_2 \in \mathcal{L}_F(\mathcal{G}) \\ \wedge\; a \in \Sigma \;\wedge\; s_1.s_2.\alpha \in \mathcal{L}_N(\mathcal{G}) \;\wedge\; P_{\Sigma_o}(s_2) = \epsilon \tag{2.9}$$

Under these assumptions, at most one transition from Q_F to Q_N or from Q_N to Q_F can take place between two visible events. Note that detecting whether an LTS has vanishing faults (resp.repairs) can be done in NLOGSPACE, and in linear time w.r.t. the size of $\mathcal{G}$.

2.3.4 A T-diagnosability test

One can consider the converse of (2.1). Specifically, $\mathcal{G}$ is *not T-diagnosable* iff

$$\exists s_1 \in \mathcal{L}_F^{min}(\mathcal{G}),\; \forall n \in \mathbb{N},\; \exists s_1.s_2 \in \mathcal{L}(\mathcal{G}) : |P_{\Sigma_o}(s_2)| \geq n, \\ \forall s_2' \leq s_2,\;\; s_1 \rightarrow s_1.s_2' \notin \mathcal{L}_F(\mathcal{G}) \;\vee\; [\![s_1 s_2']\!]_{\Sigma_o} \not\subseteq \mathcal{L}_F(\mathcal{G}) \tag{2.10}$$

In words, $\mathcal{G}$ is not T-diagnosable whenever it is possible to find a minimal faulty sequence s_1 and arbitrarily long extensions s_2 such that along the longest faulty prefix $s_2' \leq s_2$ of s_2, the detection of the fault can not occur in a timely way, either because repair occurs before any possible detection, or because the extension remains ambiguous.

It is worth noticing that the twin-machine idea used to check the diagnosability of permanent faults is not sufficient to check the T-diagnosability of repairable faults. The main obstacle is that T-diagnosability can not be characterized by pairs of equivalent runs. It is rather a global property on classes of equivalent runs in $\mathcal{G}$. This is illustrated in Fig. 2.23, where unobservable transitions are depicted as dashed arrows ($\Sigma_o = \{a\}$). This automaton is not T-diagnosable. However, by checking only *pairs* of equivalent runs, one always finds a time where ambiguity seems to vanish. For example, considering only the top and central loops, a^{3n+1} seem to be detection times for the faults that appear in these runs. To reveal that T-diagnosability does not hold, one would have to check triples of equivalent runs here. And it is easy to design examples where triples are not sufficient and one needs to escalate to quadruples of equivalent runs to reveal the non T-diagnosability, etc. This suggests a non polynomial complexity of the T-diagnosability test.

The idea of the twin-machine construction is to check whether a faulty run can create an ambiguity that can never be resolved. For repairable faults, this ambiguity signal can be directly derived from $Det_{\Sigma_o}(\mathcal{G})$, the diagnoser of $\mathcal{G}$. Consider the (deterministic) LTS $\mathcal{A} = \mathcal{G} \times Det_{\Sigma_o}(\mathcal{G})$. $Det_{\Sigma_o}(\mathcal{G})$ is a deterministic automaton over alphabet $\Sigma_o \subseteq \Sigma$, and $\mathcal{L}(Det_{\Sigma_o}(\mathcal{G})) = Traces_{\Sigma_o}(\mathcal{G})$. So $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G})$: the construction of $\mathcal{A}$ performs a simple state augmentation on $\mathcal{G}$, without changing its behavior (just like the memory automaton mentioned above). This state augmentation attaches an ambiguity status to each state of $\mathcal{G}$ as follows. States of $\mathcal{A}$ take the form $(q, x) \in Q \times \mathcal{X}$ where $\mathcal{X} = 2^Q$. So

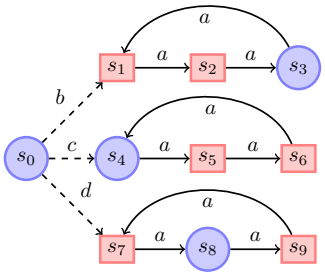


Figure 2.23: This system is not T-diagnosable, but this is not apparent if only pairs of equivalent runs are considered.

they can be labeled by elements in $\{N, F\} \times \{N, U, F\}$: for example (q, x) is of type (N, U) iff $q \in Q_N$ and x is uncertain. $\mathcal{L}_N(\mathcal{G})$ and $\mathcal{L}_F(\mathcal{G})$ are easily identifiable in $\mathcal{A}$ as sequences terminating in a state of type $(N, .)$ or $(F, .)$ respectively. A state (q, x) is said to be *minimally faulty* if and only if q is the terminal state of a sequence $s \in \mathcal{L}_F^{min}(\mathcal{G})$.

Theorem 2.4 *With the notation above, $\mathcal{G}$ is not T-diagnosable if and only if there exists a reachable minimally faulty state $(q, x) \in Q \times \mathcal{X}$ in $\mathcal{A}$ such that (q, x) is of type (F, N) or (F, U) and either*

1. *there exists a state (q', x') of type (N, N) or (N, U)*
2. *or there exists a cycle of (F, U) states*

that is reachable from (q, x) through a (possibly empty) sequence of (F, N) states followed by a sequence of (F, U) states.

Moreover, we can derive from the above Theorem the following complexity for the T-Diagnosability Problem.

Theorem 2.5 *Deciding whether an automaton $\mathcal{G}$ is T-diagnosable is a PSPACE-complete problem.*

Intuitively, the fact that T-diagnosability belongs to PSPACE is due to Theorem 2.4, whereas the hardness of the problem can be shown by reduction from the language inclusion problem, which is known to be PSPACE-complete [Koz77] (recall that this problem can be formulated as follows: given $\mathcal{G}_1, ...\mathcal{G}_n$ some deterministic finite automata, $\bigcap_{i \in 1..n} \mathcal{L}(\mathcal{G}_i) = \emptyset$?).

Example 2.9 *Back to the example depicted in Figure 2.23, the LTS $\mathcal{A} = \mathcal{G} \times Det_{\Sigma_o}(\mathcal{G})$ is given in Figure 2.24 (left-hand side), whereas its abstract view carrying only labels of composite states is given in Figure 2.24 (right-hand side) (recall that $\{b, c, d\}$ are unobservable). It is easy to check that $\mathcal{A}$ does not fulfill the conditions of Theorem 2.4. Indeed, $\mathcal{A}$ contains a (F, U) state, from which a (N, U) state is reachable (this is highlighted in the figure by a dashed arrow). Thus, as already mentioned, $\mathcal{G}$ is not T-diagnosable.* $\diamond$

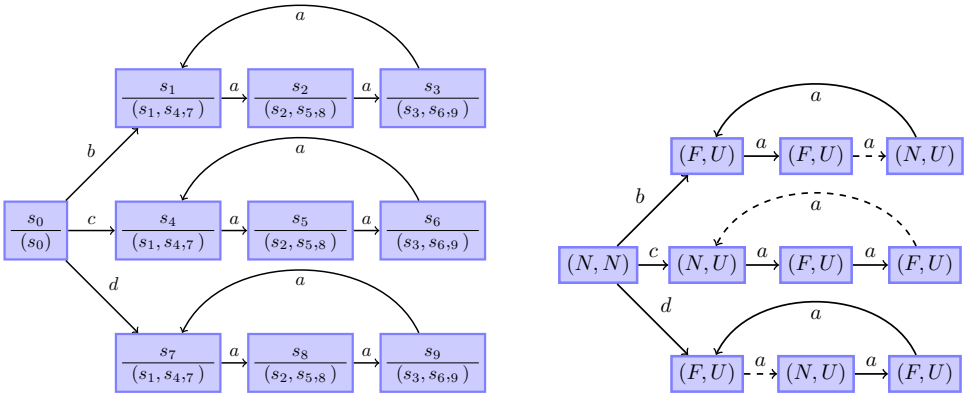


Figure 2.24: $\mathcal{A} = \mathcal{G} \times Det_{\Sigma_o}(\mathcal{G})$ and its sbstract view

2.3.5 Counting Faults

As faults are not permanent, counting the number of faults occurring at run-time is a useful information: even if a system is able to repair all occurrences of faults, a too large number of faults may indicate a major failure. To count faults, an immediate idea is to maintain a fault counter that is incremented each time the diagnoser goes from N to F and from U to F. Even if a diagnosis can be triggered in time, i.e. before the fault is repaired, T-diagnosability is not sufficient to correctly count faults along a trajectory. Fig. 2.20 reveals that this can not work as counting moves of the diagnoser from $\{N, U\}$ to F in this example would detect two faults, while v has only one fault and v'' has two. Conversely, counting only moves from N to F or from U to F leads to minoring the real number of faults that occurred in some runs.

We will say that $\mathcal{G}$ is T-Diagnosable w.r.t. N if repairs can be faithfully detected. Intuitively, this property can be checked by inversion of safe and faulty states, an then checking T-diagnosability of the so-obtained system. Consider the Diagnosis function $\mathtt{Diag}_\Omega : \mathit{Traces}_{\Sigma_o}(\mathcal{G}) \to \{N, F, U\}$ defined by (2.5). We define the function $\sharp^F_{\mathtt{Diag}_\Omega}$ from $\mathit{Traces}_{\Sigma_o}(\mathcal{G})$ to $\mathbb{N}$ as follows: Let $\mu \in \mathit{Traces}_{\Sigma_o}(\mathcal{G})$ and $\rho \in (N + U + F)^*$ the associated sequence of verdicts emitted by $\mathtt{Diag}_\Omega$. Let $\rho' \in (N + F)^*$ be the projection of ρ on the verdicts $\{N, F\}$, then $\sharp^F_{\mathtt{Diag}_\Omega}(\mu)$ is the number of occurrences of pairs NF that appear in ρ'. Intuitively, $\sharp^F_{\mathtt{Diag}_\Omega}$ is a function that will be used to count the number of faults the diagnoser is able to detect. We can define similarly the function $\sharp^N_{\mathtt{Diag}_\Omega}$, counting the number of detected repairs, by inverting N and F in the previous definition.

Given a sequence u of $\mathcal{G}$, $\sharp^F_{\mathcal{G}}(u)$ denotes the number of times $\mathcal{G}$ moves from a normal state to a faulty state in u and $\sharp^N_{\mathcal{G}}(\sigma(u))$ denotes the number of times $\mathcal{G}$ evolves from a faulty state to a normal state in u. We can now state the following proposition:

Proposition 2.7 If $\mathcal{G}$ is T-Diagnosable w.r.t. F and T-Diagnosable w.r.t. N, and has no vanishing faults nor repairs, then $\forall v \in \mathcal{L}(\mathcal{G})$ and $\mu = \Pi(v)$, then

- $0 \leq \sharp^F_{\mathcal{G}}(v) - \sharp^F_{\mathtt{Diag}_\Omega}(\mu) \leq 1$.

- $0 \leq \sharp^N_{\mathcal{G}}(v) - \sharp^N_{\mathtt{Diag}_\Omega}(\mu) \leq 1$.

Moreover if $\mathtt{Diag}_\Omega(\mu) = F$ then $\sharp^F_{\mathcal{G}}(v) = \sharp_{\mathtt{Diag}_\Omega}(\mu)$ and if $\mathtt{Diag}_\Omega(\mu) = N$ then $\sharp^N_{\mathcal{G}}(v) = \sharp^N_{\mathtt{Diag}_\Omega}(\mu)$.

Intuitively, this proposition states that we can build from the diagnoser a function that counts the number of times the system becomes faulty (resp. is repaired) with an error of at most 1. Furthermore, the difference is null as soon as the fault (resp repair) is diagnosed by the diagnoser. Note that the condition for counting in proposition 2.7 is sufficient, but not necessary as shown by the automaton of Figure 2.25.

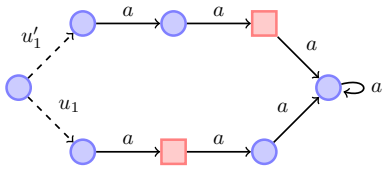


Figure 2.25: A T-Diagnosable automaton wr.t. N but not w.r.t. F

In this example, the automaton is T-diagnosable w.r.t. N but not w.r.t. F, moreover the sequence of verdicts emitted by Δ is $NUUN$. However, after reading aa we know for sure that a single fault occurred.

2.3.6 Related work

The diagnosis of such transient faults has been considered in [CLT04], which proposed four notions of diagnosability. One of them ("O-diagnosability") consists in detecting the occurrence of a transient fault, even after it has been repaired, which amounts to saturating $\mathcal{L}_F(\mathcal{G})$. Symmetrically, the "I-diagnosability" aims at detecting the occurrence of a repair, even if fault(s) followed, which amounts to inverting the roles of S_F and S_N, or to saturating the safe language $\mathcal{L}_N(\mathcal{G})$. Both notions thus match the standard (or historical) notion of diagnosability for a slightly modified version of $\mathcal{G}$.

In [JKG03], two definitions involving multiple occurrences of faults are given. A system is $K-$diagnosable if the execution of any state-trace containing at least K failures can be deduced within a finite delay from the observed behavior. K-diagnosability is not monotonic, and the authors also introduce $[1 \cdots K]$-diagnosability, that is met by systems that are J-diagnosable for every $1 \leq J \leq K$. Compared with $[1 \cdots K]$-diagnosability or simply K diagnosability, we introduced a sufficient condition under which it is possible to count exactly the number of faults that occurred in the system. Furthermore, similarly to [CLT04], the definitions of diagnosability introduced in [JKG03], do not require the detection of the fault before its repair.

In the same manner, [CLT04] introduced the notions of "P-diagnosability" and "R-diagnosability". These two notions are dual: P-diagnosability states that after the occurrence of a fault, it is always possible to detect the fact that the system is currently faulty, based on the observation (even though the fault has been repaired in the past). Conversely, $R-$diagnosability states that after a fault is repaired, it is possible to detect in finite time

whether the system is currently in a safe state. As we are mainly detecting fault occurrences, our work should only be compared to the notion of "P-diagnosability". Our notion of T-diagnosability is then stronger than P-diagnosability, as we require that detection of faults occur *before they are repaired*. It is then easy to show that whenever a system is T-diagnosable then it is also P-diagnosable. Finally, note that that deciding whether an automaton A is P-diagnosable is a also a PSPACE-complete problem (the techniques we used to prove this result are similar to the ones for the T-diagnosability problem).

2.3.7 Conclusion

We have proposed a notion of "timely-diagnosability" that requires the detection (in bounded time) of transient faults after they occur, and before they are repaired. This notion was defined for a deterministic partially observed automaton. While this choice allows one to express most properties in terms of faulty and safe languages, it leads to quite complicated criteria for T-diagnosability, as in Theorem 2.4. It could be interesting to define T-diagnosability for non-deterministic LTSs, and to explore whether criteria simplify. For example, it is likely that in the absence of vanishing faults and of vanishing repairs, T-diagnosability is preserved by unobservable-closure. Also, while the T-diagnosability of faults relies on a complicated criterion, it is likely that systems which are both T-diagnosable for faults and for repairs are more easier characterized. This subclass is quite interesting, as it corresponds to systems where all changes of state class are detected in bounded time, and in any case before they change again. So ambiguity, when it appears, can not last forever.

2.4 General Conclusion and Futur Work

In this chapter, we have been interested in the fault diagnosis of discrete event systems. In the worry of exposing a fairly general framework for diagnosis issues, the Diagnosis Problem is presented in a rather denotational spirit, as opposed to the operational spirit we find in the literature: we put the emphasis on the diagnosis function $\texttt{Predict}_\Omega$ with its correctness and boundedness diagnosability property. Correctness is an essential property that ensures the accuracy of the diagnosis. Moreover, verifying the diagnosability property of the system with respect to the supervision pattern guarantees that when using $\texttt{Predict}_\Omega$ online, an occurrence of the pattern will eventually be diagnosed, and that this eventuality can be quantified. It is the standard notion of "Diagnosability", but seen here as a mere mean to achieve a satisfactory diagnosis function; we are aware that this point of view differs from other classical approaches. The definition of diagnosability as proposed here is automata-based, with $\mathcal{G}$ and Ω, but could as well be expressed in a language-based framework. We further extended this notion of diagnosis to the notion of predictability of supervision pattern following the same framework. We finally examined the case of transient faults, that can appear and be repaired. Diagnosability in this setting means that the occurrence of a fault should always be detected in bounded time, but also before the fault is repaired. We presented this in a state-based framework but the extension to faults represented by supervision patterns is trivial as mentioned in Remark 2.1.

The previous results can be extended in several directions:

- Even-tough the decentralized approach for diagnosis of DES has been widely studied in the last past years [DLT00b, PC05, Tri01], The modular approach of diagnosis has received so far little attention. Roughly speaking, it consists in considering a systems composed of several sub-systems, that can become faulty. The idea is to build local diagnosers (one for each sub-system) and to find necessary and sufficient conditions under which the occurrence of a fault occurring in a sub-system can be surely diagnosed. This problen has benn tackled for concurrent systems with the assumption that the common events are observable [CLT06] or not necessarily observable [YP10]. Meanwhile none of them considers general patterns of faults (the fault is always localised in a single component). Extending the supervision pattern diagnosis problem to concurrent systems would thus be an interesting extension. Our view is that those diagnosers have to be built without building the whole system state space, but from (abstractions of) local diagnosers artifacts by taking into account (abstraction of) the interaction between the different components[4]. One can also think to add communications between the local diagnosers that would help them to refine their knowledge and to take local decisions regarding the presence or the absence of faults. To solve this point, we might consider the know-how and techniques from the decentralized diagnosis theory [RV01].

- An other interesting problem in this area concerns the active diagnosis which consists in controlling the system in order to render it diagnosable. Different approaches have been proposed in the literature for qualitative or quantitative active diagnosis. Recently with L. Hélouet, we started to investigate this problem from another point of view. Instead of controlling the system, we assume that the diagnoser has the ability to ask for a test that will allow to partially disambiguate the set of configurations the systems might be in. Depending of this test, the energy (or cost) to pay is different. The diagnoser starts with a certain energy provision and the active diagnosis problem is then to derive strategies that would render the system diagnosable with the help of the tests, without exhausting its energy.

- Finally, in order to consider the fault diagnosis of more realistic systems, it would be interesting to consider to be abe to includ data in the model of system. We already have results for this kind of models for automatic test generation [J15] or controller synthesis [J10] and it might be interesting to use these techniques to derive diagnosers and test for the diagnosability of those systems.

[4]This is actually an approach that we have considered for the control of concurrent systems. See Chapter 3, Section 3.2 for more details

Chapter 3

Control of Discrete Event Systems

Nowadays systems, such as embedded systems, transportation systems or energy services are becoming more and more complex. Usually, these systems are composed of several sub-systems that interact with each others either synchronously or asynchronously. Due to the interaction between these systems, some undesired behaviours might happen (e.g., mutual exclusion between configurations reached by these systems). The complexity of these systems makes that it might be difficult to manually compute a controller that avoids these "bad" behaviours. To alleviate this problem, the theory control of Discrete Event Systems [Won03, CL08] has been introduced in the eighties. The idea is the following: given a model of the system (that is supposed to accurately represent the behaviour of the system[1] and a legal behaviour, a controller has to be derived such that the resulting behaviour of the closed-loop system is included in the legal one [Won03]. As usual in the controller synthesis setting, the actions of the system can be partitioned into those that can not be controlled (e.g. inputs received from sensors, failures) and those that van constrained, by e.g., a discrete controller (typically the starting of a task). The idea of the controller is then to observe the current behaviour of the system and to say which controllable event can be triggered after this observation. One main particularity of the supervisory control problem is that the aim is to synthesize a controller which is as less restrictive as possible, meaning that the occurrence of an event is disable only when it is necessary. Moreover, as for the diagnosis problem tackled in Chapter 2, mostly due to economic point of view, not all but only a part of these events can be observed by a controller that has to take its control decision based on this partial observation.

In this chapter, we shall consider concurrent/distributed systems and tackle the supervisory control problem for this kind of systems. Roughly speaking, in this framework, two classes of systems are generally considered, depending on whether the communication between sub-systems is *synchronous*[2] or not. When the network communication can be done through multiplexing or when the synchrony hypothesis [BG92] can be made, the *decentralized control problem* and the *modular control problem* address the design

[1] When dealing with several sub-systems, we also assume that a correct model of each of them is available.

[2] By synchronous communication, we mean that the communication between controller(s) and system(s) is instantaneous.

of coordinated controllers that jointly ensure the desired properties for this kind of systems [YL00, RW92, Ric00, KvS05]. When considering *asynchronous* distributed systems, the communication delays between the components of the system must also be taken into account. Note that in both cases the *distributed/decentralized control synthesis* problem is undecidable [PR90, Tri04].

The content of this chapter is organized as follows:

- Section 3.1 recalls the bases of the Supervisory Control Problem.
- In Section 3.2, we tackle the control problem of concurrent systems, i.e., systems that communicate in a synchronous way. The idea is to computes controllers locally without having to compute the global system (that might be impossible to build due to the state-space explosion induced by the parallel composition of sub-systems). We adopted a language-based approach and proposed novel notions and algorithms in order to solve this problem [J14].
- In Section 3.3, we are going one step ahead and consider distributed systems, i.e., systems that are composed of several systems that communicate asynchronously by means of FIFO channel that are assumed to be unbounded. The aim is to build local controllers (which only have a view of their own sub-systems) that restrict the behavior of a distributed system in order to satisfy a global state avoidance property [J6], [C12]. To refine their control policy, controllers can use the FIFO queues to communicate by piggybacking extra information to the messages sent by the subsystems [C13]. We define synthesis algorithms allowing to compute the local controllers and ensure termination by using abstract interpretation techniques, to overapproximate queue contents by *regular languages* [J6].

3.1 Brief overview of the controller synthesis theory

This section briefly recalls the basic notions of the supervisory control problem that are necessary to tackle the control problem introduced in Section 3.2 and 3.3 as well as in Chapter 4. A global and complete overview of this theory can be found in [CL08].

Given a prefix-closed behavior $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G}) \subseteq \Sigma^*$ expected from the system $\mathcal{G}$, the goal of supervisory control is to enforce this behavior on $\mathcal{G}$ by pairing this system with a monitor (also called controller) that observes a subset Σ_m of the events in Σ and controls a subset Σ_c of the events in Σ, i.e. enables or disables each instance of these controllable events. $\Sigma_{uc} = \Sigma \setminus \Sigma_c$ is the set of uncontrollable events. As in diagnosis, not all but only a part of the events can be observed due to the limitation of the sensors attached to the systems or due to the distributed/concurrent nature of the systems where events cannot be seen by all sub-systems. We therefore decompose the alphabet of the system onto observable and unobservable events and we denote by Σ_m the set of observable events.

Formally, a controller is given by a function

$$\mathcal{C} \ : \ Traces_{\Sigma_m}(\mathcal{G}) \rightarrow \{\gamma \in 2^{\Sigma} \mid \Sigma_{uc} \subseteq \gamma\},$$

delivering the set of actions that are allowed in $\mathcal{G}$ after having observed the observation

trace $\mu \in Traces_{\Sigma_m}(\mathcal{G})$. We write $\mathcal{C}/\mathcal{G}$ for the closed-loop system depicted in Figure 3.1, consisting in the initial system $\mathcal{G}$ controlled by the controller $\mathcal{C}$.

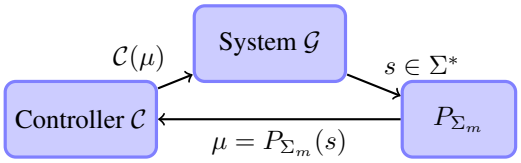


Figure 3.1: Control Architecture

The closed-loop system $\mathcal{C}/\mathcal{G}$ is a Discrete Event System that can be characterized by the language $\mathcal{L}(\mathcal{C}/\mathcal{G})$ which is recursively defined as follows:

1. $\epsilon \in \mathcal{L}(\mathcal{C}/\mathcal{G})$
2. $[(s \in \mathcal{L}(\mathcal{C}/\mathcal{G}))\ and\ s\sigma \in \mathcal{L}(\mathcal{G})\ and\ \sigma \in \mathcal{C}(P_{\Sigma_m}(s))] \Leftrightarrow s\sigma \in \mathcal{L}(\mathcal{C}/\mathcal{G})$

Remark 3.1 *Note that $\mathcal{C}$ can be seen as an LTS that acts in parallel with the system $\mathcal{G}$ as we shall see in Chapter 4*

Controllability. To be correct, a controller is not allowed to disable uncontrollable events when restricting the behavior of the system. This leads us to introduce the notion of *controllable language* [Won03].

Definition 3.1 *A prefix-closed language $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$ is controllable w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_c if*

$$\mathcal{K}.(\Sigma \setminus \Sigma_c) \cap \mathcal{L}(\mathcal{G}) \subseteq \mathcal{K}. \tag{3.1}$$

This definition states that $\mathcal{K}$ is controllable if no uncontrollable event needs to be disabled to exactly confine the system $\mathcal{L}(\mathcal{G})$ to $\mathcal{K}$.

Remark 3.2 *This definition can be extended to the case where $\mathcal{K} \not\subseteq \mathcal{L}(\mathcal{G})$ by considering $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$ as a new specification.*

Observability. Due to the fact that a controller only has a partial view of the system, its control action can change only after the occurence of an observable event. Moreover, given two sequences $s, s' \in \mathcal{L}(\mathcal{G})$ such that $s \sim_{\Sigma_m} s'$ (i.e. s and s' are observationally equivalent w.r.t. Σ_m - Cf. Def 1.4), the controller can not distinguish between these two sequences. If for some reason, the controller has to disable an event σ after the sequence s then it also has to disable it after the sequence s'. This notion is captured by the following property named *observability*

Definition 3.2 *A prefix-closed language $\mathcal{K}$ is observable w.r.t. $\mathcal{L}(\mathcal{G})$, Σ_m and Σ_c if $\forall s, s' \in \mathcal{K}$ such that $s \sim_{\Sigma_m} s'$, it entails that $s\sigma \in \mathcal{K} \Leftrightarrow s'\sigma \in \mathcal{K}$ for all $\sigma \in \Sigma_c$.*

Based on the two previous notions, we can state the following theorem:

Theorem 3.1 *Given a system $\mathcal{G}$, and a prefix-closed language $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$, there exists a controller $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}/\mathcal{G} = \mathcal{K}$ if and only if $\mathcal{K}$ is controllable w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_c and $\mathcal{K}$ is observable w.r.t. $\mathcal{L}(\mathcal{G})$, Σ_m and Σ_c.*

In general, given a system $\mathcal{G}$ and a (prefix-closed) specification $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$, $\mathcal{K}$ may be not controllable or observable, which means that it is necessary to restrict the behavior of $\mathcal{K}$ in order to obtain a sub-language of $\mathcal{K}$ that fulfills these properties. Obviously, one wants to reduce as less as possible the behavior of $\mathcal{L}(\mathcal{G})$. In the sequel we shall call a *valid controller*, a controller such that $\mathcal{L}(\mathcal{C}/\mathcal{G})$ is controllable w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_{uc} and $\mathcal{L}(\mathcal{C}/\mathcal{G})$ is observable w.r.t. $\mathcal{L}(\mathcal{G})$, Σ_m and Σ_c.
Thus, the control problem is the following:

Problem 3.1 *Given a system $\mathcal{G}$, and a prefix-closed language $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$, compute a valid controller $\mathcal{C}$ such that*

- $\mathcal{L}(\mathcal{C}/\mathcal{G}) \subseteq \mathcal{K}$
- $\mathcal{C}$ *is* maximal *in the following sense: for any other valid controller $\mathcal{C}'$, such that $\mathcal{L}(\mathcal{C}'/\mathcal{G}) \subseteq \mathcal{K}$, we have $\mathcal{L}(\mathcal{C}'/\mathcal{G}) \subseteq \mathcal{L}(\mathcal{C}/\mathcal{G})$.*

In the sequel, we shall be more interested in computing $\mathcal{L}(\mathcal{C}/\mathcal{G})$ rather than $\mathcal{C}$ itself knowing that $\mathcal{C}$ can be easily extracted from $\mathcal{L}(\mathcal{C}/\mathcal{G})$. First one can notice that the union of an arbitrary number of controllable languages is controllable, which means that there exists a unique supremal controllable language of $\mathcal{K}$. It is given by

$$\text{SupC}(\mathcal{K}, \Sigma_{uc}, \mathcal{L}(\mathcal{G})) = \mathcal{K}^{\uparrow c} = \mathcal{K} \setminus [(\mathcal{L}(\mathcal{G}) \setminus \mathcal{K})/\Sigma_{uc}^*]\Sigma^* \tag{3.2}$$

However, in general, observability is not stable under union of languages. This induces that if $\mathcal{K}$ is not observable, then in general there does not exits a supremal controllable and observable sub-language of $\mathcal{K}$. To alleviate this problem, the notion of normality has been introduced [KGS91].

Definition 3.3 *Assuming that $\Sigma_c \subseteq \Sigma_m$, a prefix-closed language $\mathcal{K}$ is* normal *w.r.t. $\mathcal{L}(\mathcal{G})$, Σ_m if $P_{\Sigma_m}^{-1}[P_{\Sigma_m}(\mathcal{K})] \cap \mathcal{L}(\mathcal{G}) \subseteq \mathcal{K}$*

Normality is stable under union of languages. Moreover, under the assumption $\Sigma_c \subseteq \Sigma_m$, normality and observability are equivalent [CL08]. Therefore, under this assumption, both controllability and observability are stable under union of languages, and there exists a supremal controllable and observable prefix-closed sub-language of $\mathcal{K}$, that we denote

$$SupCo(\mathcal{K}, \mathcal{L}(\mathcal{G}), \Sigma_c, \Sigma_m) \tag{3.3}$$

which represents the largest behavior in $\mathcal{K}$ that can be enforced by control.

Actual computation of the controlled system. Let $\mathcal{G} = (\Sigma, Q, q_o, \delta)$ be the system under consideration and $A_{\mathcal{K}} = (\Sigma, Q_{\mathcal{K}}, q_o^{\mathcal{K}}, \delta_{\mathcal{K}})$ be such that $\mathcal{L}(A_{\mathcal{K}}) = \mathcal{K}$. The algorithm computing the supremal solution is the following

(i) Compute $\mathcal{G}_\mathcal{K} = \mathcal{G} \parallel \mathcal{K}$ and consider the set of states Bad s.t.

$$Bad = \{(q, q_\mathcal{K}) \in Q \times Q_\mathcal{K} \mid \exists \sigma \in \Sigma_{uc},\ \delta_\mathcal{G}(q,\sigma)! \wedge \delta_{\mathcal{G}_\mathcal{K}}((q,q_k),\sigma) \text{ is not defined}\}$$

(these states violate the controllability property)

(ii) Compute $A = Det_{\Sigma_m}(\mathcal{G}_\mathcal{K}) = (\Sigma_m, X, x_o, \delta_A)$, with $X = 2^{Q \times Q_\mathcal{K}}$ and let

$$F = \{x \in X \mid x \cap Bad \neq \emptyset\}$$

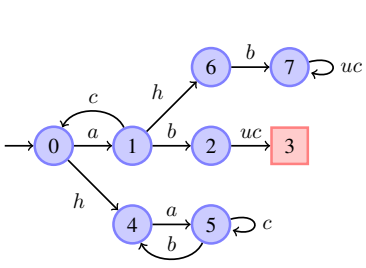
(this corresponds to the set of states that have to be removed to fulfill both controllability and observability)

(iii) $A_C = (\Sigma_m, X \setminus Coreach_{uc}(F), x_o, \delta_A)$

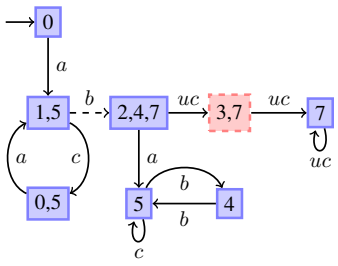
(iv) $\mathcal{G}_C = \mathcal{G} \parallel A_C$

If $\mathcal{G}_C$ is not reduced to the empty automaton, then $\mathcal{L}(\mathcal{G}_C) = \mathcal{L}(\mathcal{C}/\mathcal{G})$.

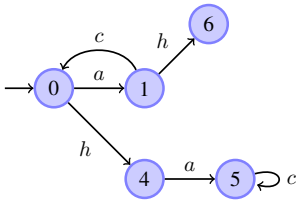
Example 3.1 *Figure 3.2(a) describes the product $\mathcal{G} \times A_\mathcal{K}$ in which the state 3 is marked. Hence, a sequence that reaches the state 3 is violating the safety property modeled by $\mathcal{K}$. We assume that $\Sigma_m = \{a, b, c, uc\}$ and that $\Sigma_c = \{a, b, c\}$. The controller decisions are performed according to the observed behavior of the system (depicted in Figure 3.2(b)). If the controller observes a sequence in $a.(c.a)^*.b.uc$, then he knows that the system is either in state 3 or 7. Thus, in order to avoid the state* $\boxed{3,7}$*, the controller needs to disable the*



(a) $\mathcal{G}$



(b) $det_{\Sigma_m}(\mathcal{G}) \times A_\mathcal{K}$



(c) The controlled system $\mathcal{C}/\mathcal{G}$

event b (uc being uncontrollable). The obtained LTS (only keeping the reachable part) is the maximal controller such that $\mathcal{L}(\mathcal{C}/\mathcal{G}) \subseteq \mathcal{K}$. The behavior of the controlled system is given in Figure 3.2(c). ◇

3.2 Control of concurrent discrete event systems

In this section, we focus on the control of Concurrent Discrete Event Systems (CDES) defined by a collection of components that interact with each other. In many applications (as e.g. manufacturing systems, control-command systems, protocol networks, etc) and control problems, systems are often modeled by several components acting concurrently. In this section, we are concerned with the control of a system where the construction of its model is assumed not to be feasible (due to the state space explosion resulting from the composition), making the use of classical supervisory control methodologies impractical.

Several approaches have been investigated to deal with the complexity issue of the control of CDES. Given a CDES $\mathcal{G}$ modeled by a set of sub-systems $\{\mathcal{G}_i\}_{1\leq i\leq n}$ (i.e., $\mathcal{G} = \|_{1\leq i\leq n}\mathcal{G}_i$) and a specification expressed by a language $\mathcal{K}$, the problem under consideration is to compute the supremal controllable sublanguage of $\mathcal{K}\cap\mathcal{L}(\mathcal{G})$ w.r.t. $\mathcal{L}(\mathcal{G})$ without having to explicitly build $\mathcal{L}(\mathcal{G})$. One classical approach consists in using Binary Decision Diagrams (BDD) [Bry92] to encode the system ([HWT92, Gun97, MW06, VLF04],[J18]) and to use transformer predicates to perform all the operations. Even though the whole system needs to be built, such encoding is efficient in the sense that it avoids the state space enumeration during the synthesis phase. Meanwhile, it is still interesting to combine these symbolic approaches with methods that are independent of the implementation. These methods should be based on the structure of a CDES and have to be efficient even though the tool, that is used to perform the supervisor synthesis, is based on enumerative methods. In [dC00b, DC00a], the authors consider the control of a product plant (i.e. systems composed of sub-systems, not sharing common events)[3]. Given a set of specifications, for each of them, a local sub-system is built from the components that are coordinated by the corresponding specification (i.e. all the components that share some events used to express it). It is then sufficient to compute local supervisors ensuring each specification with respect to the corresponding local sub-system in order to obtain the result on the whole system. An incremental and modular approach has been presented in [BMD00, AFF02]. When several specifications are under considerations, the global supervisor can be obtained by performing the parallel composition of the different corresponding "local" supervisors. Finally, closely related to the decentralized theory[4], under the hypothesis that the specification is *separable* and that the shared events are controllable, the authors of [WH91] provide a solution allowing to compute local supervisors $\mathcal{C}_i$ acting upon each sub-system $\mathcal{G}_i$ and to operate the individually controlled system $\mathcal{C}_i/\mathcal{G}_i$ concurrently in such a way that the behavior of the controlled system (i.e. $\mathcal{L}(\|_i\ \mathcal{C}_i/\mathcal{G}_i)$) corresponds to the supremal controllable sublanguage of $\mathcal{K}$ w.r.t. the system $\mathcal{L}(\mathcal{G})$. Note that each supervisor is efficiently computed since it is derived from each sub-system thus avoiding to build the whole system. The same methodology has been used in [RL03] for the control of concurrent systems for which the various components are supposed to have an identical structure. Knowing that the local controllers $\mathcal{C}_i$ are only operating on a subset of the local events, the authors give necessary and sufficient conditions over the specification $\mathcal{K}$, to obtain a non-blocking controlled system that exactly matches $\mathcal{K}$. Finally note that a decentralized architecture could be used to

[3] Given a CDES $\mathcal{G}$ modeled by $\{\mathcal{G}_i\}_{1\leq i\leq n}$, the authors actually collapse the sub-systems in order to obtain a product plant.

[4] see e.g. [CDFV88, RW92, YL00] for details related to this theory.

efficiently implement the controllers computed according to the theory of [dC00b, DC00a]. See also [JK00, LK02] and [MW06, AW02, LLW05][C27] for other works related to the control of concurrent systems and hierarchical systems.

In this section, our motivation is similar to that of the previous works. We want to use the particular structure of a concurrent system in order to avoid the building of the whole system. In this study, we have chosen not to consider the non-blocking aspect, but rather to focus on the computation of controllers for prefix-closed specifications with the aim of extending the class of specifications for which a maximal controllable solution exists. Indeed, besides the non-blocking aspect, that is tackled by some of the above works, one common condition that is required is the separability of the specification (as e.g. in [WH91]) or of the solution (as e.g. in [RL03, JK00]). However, this request happens to be quite restrictive. Indeed, this condition does not permit to model a particular behavioral interleaving between different components or a particular scheduling of actions that belong to different components (for example, it is not possible to specify that one sub-system $\mathcal{G}_1$ can only trigger a local action a_1 when one other sub-system $\mathcal{G}_2$ already triggered a local action a_2). So, our aim was to find another methodology for which the separability condition is not required. To do so, we have chosen not to adopt a "*decentralized*" approach as in [dC00b, WH91, RL03, JK00]. Instead of having one local controller per sub-system (or a set of sub-systems [dC00b]) that enforces local control actions with respect to the events of this component (resp. set of components), we perform the control on some approximations of the system, each of them derived from the behavior of one component. The behavior of these approximations is restricted so that they respect a new language property for discrete event systems called *partial controllability condition* that depends on the specification. It is shown that, a controller can be efficiently derived from these "controlled approximations" so that the behavior of the controlled system is actually controllable with respect to the system and the uncontrollable events, even though the specification is not separable. At this point we only require that the components that share an event agree on the controllable status of this event. However, it is generally the case, that the obtained solution is not the most permissive one. We thus give some new conditions under which the behavior of the obtained controlled system corresponds to the supremal controllable language contained in the specification with respect to the system. One condition concerns the system itself and requires the shared events to be controllable (as in [WH91]), the other one concerns the specification and is called local consistency (it is shown to be strictly less restrictive than separability condition).

3.2.1 Control problem formulation & Related works

We here consider a system composed of several components, sharing common events, i.e. a system $\mathcal{G}$ is modeled as a collection of LTS $\mathcal{G}_i = (Q_i, q_{oi}, \Sigma_i, \longrightarrow_i)$. The global behavior of the system is given by $\mathcal{G} = \mathcal{G}_1 \parallel \cdots \parallel \mathcal{G}_n$. We then have

$$\mathcal{L}(\mathcal{G}) = P_{\Sigma_1}^{-1}(\mathcal{L}(\mathcal{G}_1)) \cap \cdots \cap P_{\Sigma_n}^{-1}(\mathcal{L}(\mathcal{G}_n)).$$

Given a set of LTSs $(\mathcal{G}_i)_{i\leq n}$ modeling $\mathcal{G}$, we denote by Σ_s the set of shared events of $\mathcal{G}$. It represents the set of events shared by at least two different sub-systems:

$$\Sigma_s = \bigcup_{i\neq j}(\Sigma_i \cap \Sigma_j)$$

The alphabet of one sub-system $\mathcal{G}_i$ is split into the controllable event set $\Sigma_{i,c}$ and the uncontrollable event set $\Sigma_{i,uc}$, i.e. $\Sigma_i = \Sigma_{i,uc} \dot\cup \Sigma_{i,c}$. The alphabet of the global system $\mathcal{G}$ is given by:

$$\Sigma = \bigcup_i \Sigma_i, \quad \Sigma_c = \bigcup_i \Sigma_{i,c} \text{ and } \Sigma_{uc} = \Sigma \setminus \Sigma_c.$$

Moreover, we assume that the following relation holds between the control status of shared events:

$$\forall i, j,\ \Sigma_{i,uc} \cap \Sigma_{j,c} = \emptyset \tag{3.4}$$

which simply means that the components that share an event agree on the control status of this event. Under this hypothesis, we have that $\Sigma_{uc} = \cup_i \Sigma_{i,uc}$.

Let $\mathcal{K} \subseteq \Sigma^*$ be the specification. The problem we are interested in is the computation of the supremal controllable sub-language $(\mathcal{K} \cap \mathcal{L}(G))^{\uparrow c}$ of $\mathcal{K} \cap \mathcal{L}(G)$ w.r.t. Σ_{uc} and $\mathcal{L}(G)$. As we consider concurrent systems, the construction of the entire system may not be feasible (due to the state-space explosion resulting from the composition), as well as the construction of an LTS generating $\mathcal{K} \cap \mathcal{L}(G)$. It is then important to design algorithms that perform the controller synthesis phase by taking advantage of the structure of G without building it. Hence, the actual problem is to compute $(\mathcal{K} \cap \mathcal{L}(G))^{\uparrow c}$ without computing neither $\mathcal{L}(G)$ nor $\mathcal{K} \cap \mathcal{L}(G)$.

3.2.1.1 comparison between approaches

Due to the concurrent nature of the system it seems quite natural to solve the control problem using a decentralized methodology based on the structure of the system. First introduced in [WH91], this corresponds to the classical approach that has been investigated in the literature so far [dC00b, RL03, JK00, LK02][5]. However, it is worthwhile noting that a centralized approach can also be used to solve this problem. We now briefly outline the works of [WH91] based on a decentralized approach before presenting our methodology based on a centralized but modular approach and pointing out the differences between these two approaches.

A Decentralized approach The works of [WH91] are related to the decentralized control theory. The authors consider the control of Concurrent Discrete Event Systems $\mathcal{G}$ modeled by several sub-systems $\{\mathcal{G}_i\}_{1\leq i\leq n}$. Given a specification modeled as a prefix-closed language $\mathcal{K}$, they provide a method that computes local modular controllers $\mathcal{C}_i$ on $\mathcal{G}_i$ (based on a notion of separable specification (See Definition 3.4)) in such a way that language generated by the parallel composition of the local controlled systems $\mathcal{C}_i/\mathcal{G}_i$ corresponds to the supremal controllable sublanguage of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$ w.r.t. $\mathcal{L}(\mathcal{G})$.

[5] Even though differently presented, one can use a decentralized architecture to implement the result of [dC00b].

Definition 3.4 *$\mathcal{L} \subseteq \Sigma^*$ is said to be separable w.r.t. $\{\Sigma_i\}_{i\leq n}$ with $\cup_{i\leq n}\Sigma_i = \Sigma$, whenever there exists a set of languages $\{\mathcal{L}_i\}_{i\leq n}$, s.t. $\mathcal{L}_i \subseteq \Sigma_i^*$ and $\mathcal{L} = \mathcal{L}_1 \parallel \cdots \parallel \mathcal{L}_n$.*

It may be shown that when $\mathcal{L}$ is separable w.r.t. $\{\Sigma_i\}_{i\leq n}$, then this language can be rewritten as : $\mathcal{L} = P_{\Sigma_1}(\mathcal{L}) \parallel \cdots \parallel P_{\Sigma_1}(\mathcal{L})$. Now, based on Definition 3.4, the authors of [WH91] have shown that, given a concurrent system $\mathcal{G}$ modeled by $\{\mathcal{G}_i\}_{1\leq i\leq n}$ and $\mathcal{K} \subseteq \Sigma^*$, if $\Sigma_s \subseteq \Sigma_c$, then there exists a set of local controllers $(\mathcal{C}_i)_{1\leq i\leq n}$ such that $\|_{1\leq i\leq n} L(\mathcal{C}_i/\mathcal{G}_i) = \text{SupC}(\mathcal{K}\cap L(\mathcal{G}), \Sigma_{uc}, \mathcal{L}(\mathcal{G}))$ if and only if $\text{SupC}(\mathcal{K}\cap L(\mathcal{G}), \Sigma_{uc}, \mathcal{L}(\mathcal{G}))$ is separable. Even though interesting, this result requires to build $\text{SupC}(\mathcal{K} \cap L(\mathcal{G}), \Sigma_{uc}, \mathcal{L}(\mathcal{G}))$ and to check whether it is separable or not. Nevertheless, the authors have proved the following theorem that gives a condition under which no global verification is required:

Theorem 3.2 *Let $\mathcal{G}$ be a concurrent system modeled by $\{\mathcal{G}_i\}_{1\leq i\leq n}$, with $\mathcal{L}(\mathcal{G}_i) \subseteq \Sigma_i^*$ and $\mathcal{K} \subseteq \Sigma^*$ the specification. If $\Sigma_s \subseteq \Sigma_c$ and $\mathcal{K}$ is separable w.r.t. $\{\Sigma_i\}_{i\leq n}$, then*

$$\|_{i\leq n}\ SupC(P_i(\mathcal{K}) \cap \mathcal{L}(\mathcal{G}_i), \Sigma_{i,uc}, L(\mathcal{G}_i)) = SupC(\mathcal{K} \cap \mathcal{L}(\mathcal{G}), \Sigma_{uc}, \mathcal{L}(\mathcal{G})).$$

Hence, given a Concurrent DES $\mathcal{G}$ and a separable specification $\mathcal{K}$, Theorem 3.2 shows that there exists a set of local controllers $\mathcal{C}_i$ acting upon $\mathcal{G}_i$ such that the parallel composition of the local controlled systems actually corresponds to the supremal controllable sub-language of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$, i.e., $\|_{i\leq n}\ \mathcal{L}(\mathcal{C}_i/\mathcal{G}_i) = (\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$ (c.f. Fig. 3.2).

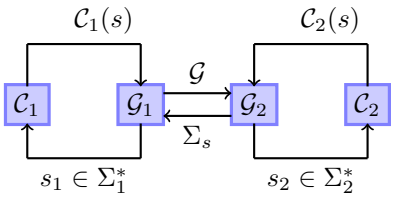


Figure 3.2: A Decentralized architecture for Concurrent DES

Complexity overview. If $\mathcal{K}$ is separable w.r.t. $\{\Sigma_i\}_{i\leq n}$ (which can be checked in $\mathcal{O}(N_{\mathcal{K}}^{n+1})$, where $N_{\mathcal{K}}$ is the size of the LTS that generates $\mathcal{K}$), then synthesizing the local controllers requires the computation of the projection of $\mathcal{K}$ over Σ_i. In the worst case, the size of the LTS that generates $P_{\Sigma_i}(\mathcal{K})$ is in $\mathcal{O}(2^{N_{\mathcal{K}}})$. Hence, in the worst case, solving the supervisory control problem requires $\mathcal{O}(n.2^{N_{\mathcal{K}}}.N + N_{\mathcal{K}}^{n+1})$ in both space and time where N is the size of each component. Note that if the specification is directly given as a concurrent specification $\mathcal{K}^1 \parallel \cdots \parallel \mathcal{K}^n$ that is compatible with the alphabet of the system then the complexity raises down to $\mathcal{O}(n.N.N_{\mathcal{K}})$, where $N_{\mathcal{K}}$ is the size of each component $\mathcal{K}^i$.

Our approach is different and is more related to the modular approach of [WR88]. Indeed, the system $\mathcal{G}$ can be described by the following parallel composition of LTS $\mathcal{G} = \|_{i\leq n}\ \mathcal{G}_i^{-1}$, where $\mathcal{G}_i^{-1}$ is the LTS such that $\mathcal{L}(\mathcal{G}_i^{-1}) = P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))$. In fact, each $\mathcal{G}_i^{-1}$ can be seen as an approximation of the system $\mathcal{G}$, which corresponds to all the knowledge of the behavior that we may deduce on $\mathcal{G}$ from $\mathcal{G}_i$ knowing that this component is

composed with other components. Compared to [WH91], we adopt a dual approach. Instead of controlling each component $\mathcal{G}_i$ (i.e. $\mathcal{L}(\mathcal{G}_i)$) to enforce $P_{\Sigma_i}(\mathcal{K})$, we have chosen to control the approximations $\mathcal{L}(\mathcal{G}_i^{-1})$ of the system in order to enforce $\mathcal{K}$ in a modular fashion. Hence our problem is to find conditions under which we are able to synthesize controllers such that

$$L(\mathcal{C}_1/P_{\Sigma_1}^{-1}(\mathcal{G}_1)) \cap \cdots \cap L(\mathcal{C}_n/P_{\Sigma_n}^{-1}(\mathcal{G}_n)) \;=\; (\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$$

Unfortunately, it is not sufficient to compute a controller $\mathcal{C}_i$ acting upon $\mathcal{G}_i^{-1}$ that restricts the behavior $\mathcal{L}(\mathcal{G}_i^{-1})$ to the supremal controllable sublanguage of $\mathcal{K} \cap \mathcal{L}(\mathcal{G}_i^{-1})$ and to operate the controlled systems $\mathcal{C}_i/\mathcal{G}_i^{-1}$ concurrently to obtain the supremal controllable sublanguage of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$.

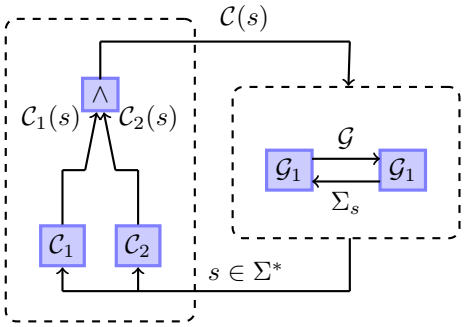


Figure 3.3: Supervision Scheme

This is basically due to the duality between the local and global controllable events. The idea is then to refine the notion of controllability in order to take into account the fact that uncontrollable events may be local to a component. The property that we ensure on each $P_{\Sigma_i}^{-1}(\mathcal{G}_i)$ according to $\mathcal{K}$ is called *the partial controllability condition* and is defined in Section 3.2.1.2.

3.2.1.2 Partial Controllability Condition

Definition 3.5 *Let $\mathcal{M} \subseteq \mathcal{L} \subseteq \Sigma^*$ be prefix-closed languages. Let $\Sigma'_{uc} \subseteq \Sigma_{uc} \subseteq \Sigma$ be two sub-alphabets of Σ. Let $\mathcal{M}' \subseteq \mathcal{M}$, then $\mathcal{M}'$ is partially controllable with respect to Σ'_{uc}, Σ_{uc}, $\mathcal{M}$ and $\mathcal{L}$ if*

(i) $\mathcal{M}'$ is controllable w.r.t Σ'_{uc} and $\mathcal{L}$.

(ii) $\mathcal{M}'$ is controllable w.r.t Σ_{uc} and $\mathcal{M}$.

Intuitively, $\mathcal{L}$ will be seen as an approximation of the system w.r.t. one of its components and $\mathcal{M}$ as the initial specification (C.f. Section 3.2.2 and Theorem 3.3). Now, given a sub-behavior of $\mathcal{M}$, the idea is that we may allow the violation of the controllability condition by triggering an uncontrollable event σ that is not local (i.e. $\sigma \in \Sigma_{uc} \setminus \Sigma'_{uc}$), because $\mathcal{L}$

only constitutes an approximation of the system and because at least one of the other controllers, computed from the other approximations, will avoid these events to be admissible. However, we still want to enforce the controllability of $\mathcal{M}'$ w.r.t. $\mathcal{M}$ and Σ_{uc} because $\mathcal{M}$ will constitute the actual specification and not an approximation. In general, $\mathcal{M}$ is not partially controllable with respect to Σ'_{uc}, Σ_{uc}, $\mathcal{M}$ and $\mathcal{L}$ (e.g. if $\mathcal{M}$ is not controllable w.r.t. Σ'_{uc} and $\mathcal{L}$). However, it can be shown that there exists a supremal sub-language of $\mathcal{M}$ that satisfies this property.

Proposition 3.1 *Let $\mathcal{M} \subseteq \mathcal{L} \subseteq \Sigma^*$ be prefix-closed languages, $\Sigma'_{uc} \subseteq \Sigma_{uc}$. There exists a unique supremal language, denoted by $\mathcal{M}^{\uparrow pc}$, which is partially controllable w.r.t Σ'_{uc}, Σ_{uc}, $\mathcal{M}$ and $\mathcal{L}$. Moreover*

$$\mathcal{M}^{\uparrow pc} = \mathit{SupC}(\mathit{SupC}(\mathcal{M}, \Sigma'_{uc}, \mathcal{L}), \Sigma_{uc}, \mathcal{M}) \tag{3.5}$$

Proposition 3.1 offers a practical way to compute $\mathcal{M}^{\uparrow pc}$, the supremal partially controllable sub-language of $\mathcal{M}$ w.r.t. to Σ'_{uc}, Σ_{uc}, $\mathcal{M}$ and $\mathcal{L}$. This language will be sometimes denoted $\text{SupPC}(\mathcal{M}, \Sigma'_{uc}, \Sigma_{uc}, \mathcal{L})$ in the sequel. The complexity for the computation of $\mathcal{M}^{\uparrow pc}$ is in $\mathcal{O}(N_{\mathcal{M}}.N_{\mathcal{L}})$, where $N_{\mathcal{M}}$ (resp $N_{\mathcal{L}}$) is the size of the LTS generating $\mathcal{M}$ (resp. $\mathcal{L}$).

3.2.2 Control of Concurrent DES

Given a Concurrent DES $\mathcal{G}$ modeled by LTS $\{\mathcal{G}_i\}_{1\leq i\leq n}$, and a specification $\mathcal{K}$, we want to compute a controllable sub-language of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$ w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_{uc}, without building $\mathcal{G}$ itself.

Modular Computation of a controllable sub-language of $\mathcal{K}$ w.r.t. $\mathcal{L}(\mathcal{G})$. Based on the concept of partial controllability applied on $\mathcal{K}$ and on the approximations of the system $P^{-1}_{\Sigma_i}(\mathcal{L}(\mathcal{G}_i))$ derived from each of its components, the next theorem provides a modular way to compute a sub-language of $\mathcal{K}$ that is controllable with respect to $\mathcal{G}$.

Theorem 3.3 *Let $\mathcal{G}$ a concurrent system modeled by LTS $\{\mathcal{G}_i\}_{1\leq i\leq n}$ and let $\mathcal{K} \subseteq \Sigma^*$ be a prefix-closed language modeling the specification. For $i \leq n$, we denote*

- *$\mathcal{K}_i = \mathcal{K} \cap P^{-1}_{\Sigma_i}(\mathcal{L}(\mathcal{G}_i))$, and*
- *$\mathcal{K}_i^{\uparrow pc}$ the supremal sublanguage of $\mathcal{K}_i$ partially controllable with respect to $\Sigma_{i,uc}$, Σ_{uc}, $\mathcal{K}_i$ and $P^{-1}_{\Sigma_i}(\mathcal{L}(\mathcal{G}_i))$.*

Then, $\mathcal{K}_1^{\uparrow pc} \cap \cdots \cap \mathcal{K}_n^{\uparrow pc}$ is controllable with respect to Σ_{uc} and $\mathcal{L}(\mathcal{G})$.

Theorem 3.3 gives us a modular method to compute a sub-language of $\mathcal{K}$ that is controllable w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_{uc}. Moreover, this computation is performed without building the system $\mathcal{G}$ (i.e. there is no need to perform the parallel composition between the components of $\mathcal{G}$). From a computational point of view, based on Section 3.2.1.2, for $i \leq n$, the computation of $\mathcal{K}_i^{\uparrow pc}$ is in $\mathcal{O}(N.N_{\mathcal{K}})$ (where N represents the number of states of any sub system $\mathcal{G}_i$ and $N_{\mathcal{K}}$ the one of the specification $\mathcal{K}$).

Example 3.2 *Let $\mathcal{G} = \mathcal{G}_1 \parallel \mathcal{G}_2$, as described in Example 1.2 with $\Sigma_{1_{uc}} = \{u_1\}$ and $\Sigma_{2_{uc}} = \{u_2\}$ and let us consider the specification given by the language $\mathcal{K} \subseteq \Sigma^*$ as described in Fig. 3.4(a)[6]. Our aim is to compute a sub-language of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$ that is controllable w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_{uc}. Following Theorem 3.3, we first compute the languages $\mathcal{K}_i = \mathcal{K} \cap P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))$, $i = 1, 2$. The LTS generating $\mathcal{K}_1$ and $\mathcal{K}_2$ are represented in Fig. 3.4(b) and 3.4(c).*

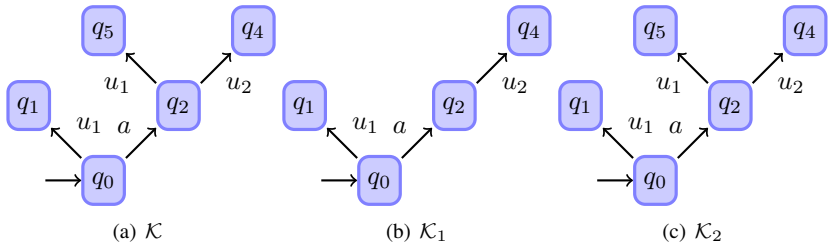


Figure 3.4: $\mathcal{K}$ and the derived specifications $\mathcal{K}_1$ and $\mathcal{K}_2$

Now based on Eq.(3.5), we compute $\mathcal{K}_1^{\uparrow pc}$ (resp $\mathcal{K}_2^{\uparrow pc}$), the supremal language of $\mathcal{K}_1$ (resp. $\mathcal{K}_2$) that is partially controllable w.r.t. Σ_{uc}, $\Sigma_{1,uc}$ and $P_{\Sigma_1}^{-1}(\mathcal{L}(\mathcal{G}_1)$, (resp $\Sigma_{2,uc}$ and $P_{\Sigma_2}^{-1}(\mathcal{L}(\mathcal{G}_2))$) (c.f. Fig. 3.5). The intersection of these two languages is the language $\mathcal{K}_1^{\uparrow pc} \cap \mathcal{K}_2^{\uparrow pc} = \{\epsilon, u_1\}$, that is obviously controllable w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_{uc}.

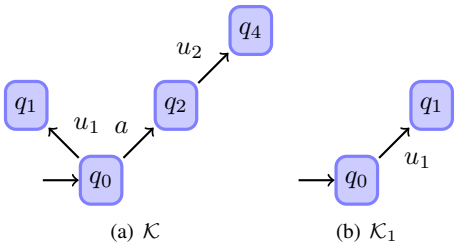


Figure 3.5: The resulting supremal partially controllable languages

Let us now describe how a controller can be extracted from the previously computed languages and how it can act upon $\mathcal{G}$ in order to achieve the specification $\mathcal{K}$. With the notations of Theorem 3.3, $\bigcap_{i \leq n} \mathcal{K}_i^{\uparrow pc}$ is controllable with respect to Σ_{uc} and $\mathcal{L}(\mathcal{G})$. However, it is not interesting to perform the intersection between these languages and to derive a controller from the result (all the computational advantages of our method would be lost).

Each $\mathcal{K}_i^{\uparrow pc}$ can be seen as a controller $\mathcal{C}_i$ which is able to restrict behaviors of $P_{\Sigma_i}^{-1}(\mathcal{G}_i)$ to a controllable one with respect to $\Sigma_{i,uc}$. Since $P_{\Sigma_i}^{-1}(\mathcal{G}_i)$ is an over-approximation of $\mathcal{G}$, it is also possible to apply $\mathcal{C}_i$ to $\mathcal{G}$. Doing so for all i leads to the concept of modularity described in [WR88]. But since the controlled systems generated by two different

[6]Note that b belongs to the alphabet of $\mathcal{K}$ even though it can not be triggered.

$\mathcal{C}_i$ are controllable with respect to two different sets of uncontrollable events, the results of [WR88] can not be used. However, Theorem 3.3 justifies the use of a modular architecture. Hence, from a given behavior, only events enabled by all the $\mathcal{C}_i$ (derived from $\mathcal{K}_i^{\uparrow pc}$) are effectively enabled. In other words, the global controller acting upon $\mathcal{G}$ is given by $\mathcal{C}(s) = \mathcal{C}_1(s) \cap \cdots \cap \mathcal{C}_n(s)$. The controller architecture is summarized in Fig. 3.3.

3.2.2.1 Computation of $(\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$

The above methodology allows us to compute a controllable sub-language of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$. Indeed, according to Theorem 3.3, we have that $\bigcap_{i \leq n} \mathcal{K}_i^{\uparrow pc}$ is a controllable sub-language of $(\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$. However, it may happen that this language is not the maximal permissive one (as in Example 1.2 in which the supremal controllable sublanguage of $\mathcal{K} \cap \mathcal{L}(\mathcal{G})$ is given by $(\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c} = \{\epsilon, a, u_1, au_2\}$). In this section, we present some conditions under which Theorem 3.3 gives access to the supremal solution.

First, it is worthwhile noting that, in general, uncontrollable shared events are not adequate to perform local computations. Indeed, in order to ensure the partial controllability of $\mathcal{K}_i$ w.r.t. $\Sigma_{i,uc}$, Σ_{uc}, $P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}))$, we may need to disable a shared uncontrollable event by control, even though this event is not fireable in the global system (this is due to the fact that we are working on approximations and thus with local informations). This leads us to restrict the class of CDES to the class that do not share uncontrollable events (i.e. $\Sigma_s \subseteq \Sigma_c$).

The specification is a subset of $\mathcal{L}(\mathcal{G})$. Theorem 3.4 shows that whenever the specification models sub-behaviors of the system (i.e $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$), then applying Theorem 3.3 provides a maximal permissive controller.

Theorem 3.4 *If $\Sigma_s \subseteq \Sigma_c$ and $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$, then with the notations of Theorem 3.3, $\bigcap_{i \leq n} \mathcal{K}_i^{\uparrow pc} = \mathcal{K}^{\uparrow c}$.*

Theorem 3.4 states that whenever the shared events are controllable and the language of the specification is included in that of the system, our method computes the supremal controllable sub-language of $\mathcal{K}$ w.r.t. $\mathcal{L}(\mathcal{G})$ and Σ_{uc}. We recall that the complexity of our method is in $\mathcal{O}(n.N.N_{\mathcal{K}})$. This has to be opposed to the complexity $\mathcal{O}(N^n.N_{\mathcal{K}})$ of computing $(\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$ when $\mathcal{G}$ is seen as a unique LTS (of course this complexity only stands for prefix-closed specification). Finally note that it is sufficient to check that $\forall 1 \leq i \leq n$, $\mathcal{K} \subseteq P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))$ in order to check that $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$. Hence, it is not necessary to compute $\mathcal{L}(\mathcal{G})$.

Example 3.3 *Let us consider the system given in Fig. 3.4 again, and the specification given by $\mathcal{K} = \overline{\{u_1, au_2\}}$. It is easy to show that $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$. Moreover, for this specification, we have $\mathcal{K} = {\mathcal{K}_1}^{\uparrow pc} = {\mathcal{K}_2}^{\uparrow pc}$, which implies that ${\mathcal{K}_1}^{\uparrow pc} \cap {\mathcal{K}_2}^{\uparrow pc} = \mathcal{K}$. Hence according to Theorem 3.4, $\mathcal{K} = SupC(\mathcal{K}, \Sigma_{uc}, \mathcal{L})$.*

The specification is locally consistent w.r.t. $\mathcal{L}(\mathcal{G})$. In some situations, modeling the specification by a language included in the language of the system may lead to a language that is too complex to be efficiently represented. Moreover, requiring the inclusion of

languages induces that the specification of $\mathcal{K}$ may be itself relatively difficult to identify as $\mathcal{L}(\mathcal{G})$ is not known (the inclusion can only be checked *a posteriori*). One may consider a specification that requires the system to trigger only once a particular event, say a. As such, this specification can be modeled by an LTS with two states. However, if we request this specification to be included in the behavior of the system, we would have to unfold the system in order to only take into account the behaviors that match this specification.

To alleviate these problems (size, inclusion and difficulty of modeling), we now introduce a new condition under which our methodology gives access to the supremal controllable sub-language of $\mathcal{K}$ w.r.t. $\mathcal{L}$ and Σ_{uc}. This condition does not require the specification to be included in the one of the system and allows us to have specifications that are relatively independent of the system. This condition is called *local consistency* and is given by Definition 3.7. But first, we introduce the notion of *consistency*:

Definition 3.6 *Let $\Sigma' \subseteq \Sigma$ be two alphabets and let $\mathcal{M} \subseteq \Sigma^*$ be a prefix-closed language. Let us consider alphabets $\Sigma_{uc} \subseteq \Sigma$ and $\Sigma'_{uc} = \Sigma_{uc} \cap \Sigma'$. $\mathcal{M}$ is consistent with respect to Σ_{uc} and $P_{\Sigma'}$ if $\forall s \in \mathcal{M},\ \forall s' \in s^{-1}(\mathcal{M}, \Sigma_{uc}), \forall \sigma \in \Sigma'_{uc}$,*

$$P_{\Sigma'}(s')\sigma \in s^{-1}(\mathcal{M}\,\Sigma'_{uc}) \Rightarrow s'\sigma \in s^{-1}(\mathcal{M}, \Sigma_{uc}). \tag{3.6}$$

Definition 3.6 captures a certain interleaving between the local (Σ'_{uc}) and global uncontrollable events. In particular, among other aspects, this condition induces that if after a sequence s of $\mathcal{M}$, there is a local uncontrollable event that is admissible, then this event is admissible whenever $\mathcal{M}$ triggers uncontrollable events that belong to $\Sigma_{uc} \setminus \Sigma'_{uc}$ (C.f. Fig. 3.6)).

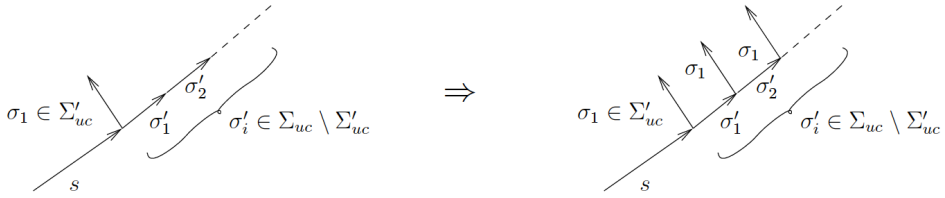
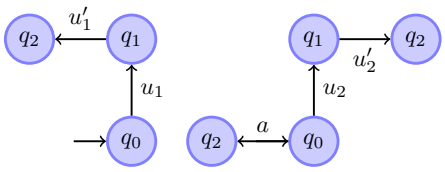


Figure 3.6: An aspect of the Definition 3.6 reflecting the interleaving.

Definition 3.7 *Let us consider a concurrent system $\mathcal{G}$ modeled by LTS $\{\mathcal{G}_i\}_{1\leq i\leq n}$ and $\mathcal{K}$ be a prefix-closed language over Σ. $\mathcal{K}$ is* locally consistent *with respect to Σ_{uc} and $\mathcal{G}$ if $\forall i \in \{1, \ldots, n\}$, $\mathcal{K} \cap P^{-1}_{\Sigma_i}(\mathcal{L}(\mathcal{G}_i))$ is consistent with respect to Σ_{uc} and P_{Σ_i}.*

Therefore, a language is locally consistent for a concurrent systems whenever it is consistent according to each local sub-system. Intuitively, if $\mathcal{K}$ is *locally consistent* with respect to Σ_{uc} and $\mathcal{L}(\mathcal{G})$, then it means that the possible interleaving between the local/global uncontrollable events w.r.t. each approximation of the system are taken into account in the specification $\mathcal{K}$. Roughly speaking, it means that $\mathcal{K}$ respects the interleaving between the local and global uncontrollable events as long as they happen in the approximations.

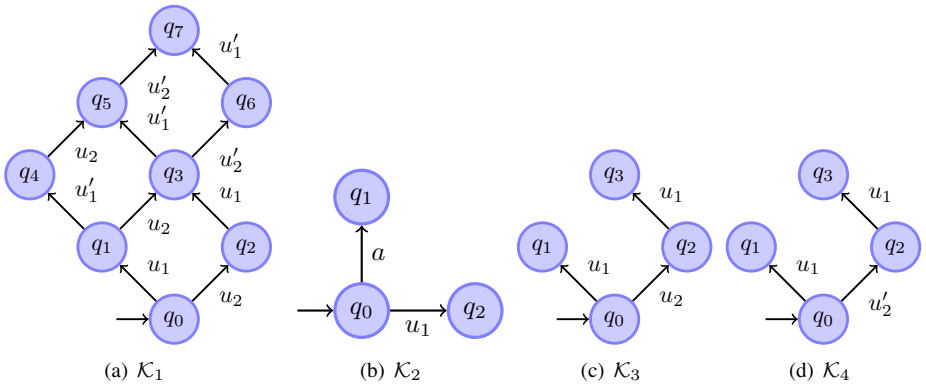
Example 3.4 *Let us consider the concurrent system given by the parallel composition of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *respectively represented in Fig. 3.7* (a) *and 3.7* (b)*. The sets of uncontrollable events are respectively given by* $\Sigma_1 = \{u_1, u_1'\}$ *and* $\Sigma_2 = \{u_2, u_2'\}$*. In this example, we*



(a) $\mathcal{G}_1$ (b) $\mathcal{G}_2$

Figure 3.7: $\mathcal{G} = \mathcal{G}_1 \| \mathcal{G}_2$.

consider several specifications, given by Fig. 3.8(a) to 3.8(d). In order to check the local consistency of $\mathcal{K}^j$ *we need to check the consistency of* $\mathcal{K}^j \cap P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))$ *w.r.t.* Σ_{uc} *and* $P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))$*.*



(a) $\mathcal{K}_1$ (b) $\mathcal{K}_2$ (c) $\mathcal{K}_3$ (d) $\mathcal{K}_4$

Figure 3.8: Different specifications

It is easy to show that both $\mathcal{K}_1$ *and* $\mathcal{K}_2$ *are locally consistent w.r.t.* $\mathcal{G}$*. However, one can check that* $\mathcal{K}^3$ *is not locally consistent because* $\mathcal{K}_3 \cap P_2^{-1}(\mathcal{L}(\mathcal{G}_2))$ *is not consistent. Indeed, in* $\mathcal{K}_3 \cap P_2^{-1}(\mathcal{L}(\mathcal{G}_2))$ $(= \mathcal{K}_3)$*, we have that* $P_{\Sigma_2}(u_1).u_2 \in \mathcal{K}_3$ *(since* $P_{\Sigma_2}(u_1).u_2 = u_2$*) while* $u_1.u_2 \notin \mathcal{K}_3$*. Finally, even-though* $\mathcal{K}_4$ *seems very similar to* $\mathcal{K}_3$*,* $\mathcal{K}_4$ *is locally consistent. Indeed,* $\mathcal{K}_4 \cap P_{\Sigma_2}^{-1}(\mathcal{L}(\mathcal{G}_2)) = \{u_1\}$ *and is consistent. The difference between* $\mathcal{K}_3$ *and* $\mathcal{K}_4$ *is that the sequence* u_2 *belongs to* $P_{\Sigma_2}^{-1}(\mathcal{L}(\mathcal{G}_2))$ *whereas* u_2' *does not.*

Theorem 3.5 *If* $\Sigma_s \subseteq \Sigma_c$ *and* $\mathcal{K}$ *is locally consistent, then with the notations of Theorem 3.3,*

$$\bigcap_{1 \leq i \leq n} \mathcal{K}_i^{\uparrow pc} = (\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$$

Theorem 3.5 states that the *local consistency* condition together with $\Sigma_s \subseteq \Sigma_c$ are sufficient

conditions under which our approach solves the *Basic Supervisory Control Problem*[7]. Now, given prefix-closed languages $\mathcal{K}$ and $\mathcal{L}(\mathcal{G})$, the complexity of checking *local consistency* is $\mathcal{O}(n.N^2.N_{\mathcal{K}}^2)$ (where $N_{\mathcal{K}}$ denotes the number of states of the LTS generating $\mathcal{K}$ and N represents the number of states of any sub system $\mathcal{G}_i$ (see [Gau04] for details regarding how the local consistency check is performed). Moreover, as previously mentioned, the complexity of computing $(\mathcal{K} \cap \mathcal{L}(\mathcal{G}))^{\uparrow c}$ with our method is $\mathcal{O}(n.N_{\mathcal{K}}.N)$. Therefore, whenever our method can be applied, its overall complexity is $\mathcal{O}(n.N^2.N_{\mathcal{K}}^2)$.

3.2.2.2 How to relax the assumption $\Sigma_s \subseteq \Sigma_c$

According to Theorems 3.4 or 3.5, in order to solve the Basic Supervisory Control Problem, we do require (1) the specification to be locally consistent or to be included in $\mathcal{L}(\mathcal{G})$ and (2) the shared events to be controllable. However, none of these conditions is necessary (only sufficient) to obtain a maximal solution. The next corollary shows that it is possible to suppress the second condition as far as the uncontrollable shared events are not involved during the computation phase. Moreover, the modular computation approach gives an efficient way to check for this. If this property holds, our approach provides a maximal solution to the BSCP, even when some shared event are uncontrollable.

Corollary 3.1 *Consider a concurrent system $\mathcal{G}$ modeled by LTS $\{\mathcal{G}_i\}_{1\leq i\leq n}$ s.t. $\mathcal{L}(\mathcal{G}_i) \subseteq \Sigma_i^*$ and $\mathcal{K} \subseteq \Sigma^*$. If $\mathcal{K} \subseteq \mathcal{L}(\mathcal{G})$ or $\mathcal{K}$ is locally consistent w.r.t $\Sigma_{uc} \setminus \Sigma_s$ and $\mathcal{G}$ and if*

$$\forall i, \textit{SupPC}(\mathcal{K}_i, \Sigma_{i,uc}, \Sigma_{uc}, P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))) = \textit{SupPC}(\mathcal{K}_i, (\Sigma_{i,uc} \backslash \Sigma_s), (\Sigma_{uc} \backslash \Sigma_s), P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))) \tag{3.7}$$

Then

$$\bigcap_i \textit{SupPC}(\mathcal{K}_i, \Sigma_{i,uc}, \Sigma_{uc}, P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i))) = \textit{SupC}(\mathcal{K} \cap L(\mathcal{G}), \Sigma_{uc}, \mathcal{L}(\mathcal{G}))$$

If Condition 3.7 holds then it simply says that uncontrollable shared events are not involved in the computation of the supremal partially controllable sub-language of $\mathcal{K}$ with respect to $\Sigma_{i,uc}$ and Σ_{uc} (i.e. considering $\Sigma_{i,uc} \cap \Sigma_s$ as controllable or not gives access to the same solution).

3.2.3 Conclusion

In this section, we investigate the Supervisory Control of Concurrent Discrete Event Systems. In particular, we propose an efficient modular method that computes the supremal controllable language included in a specification $\mathcal{K}$ w.r.t. to the system $\mathcal{G}$. This computation is performed without building the whole system, hence avoiding the state space explosion induced by the concurrent nature of the system. Moreover, if one wants to change a component of $\mathcal{G}$, e.g. replacing $\mathcal{G}_i$ by $\mathcal{G}_i'$, then as far as $\mathcal{G}_i'$ is expressed using the same alphabet as the one of G_i with the same partitioning between the controllable/uncontrollable event, then it is sufficient to recompute $\mathcal{K}_i'^{\uparrow pc} = (\mathcal{K} \cap P_{\Sigma_i}^{-1}(\mathcal{L}(\mathcal{G}_i'))^{\uparrow pc}$ in order to obtain the new controller (note that only the conditions referring to $\mathcal{G}_i'$ have to be (re)-checked). Hence this

[7] In [C28], a similar result was obtained but with a more restrictive condition named $\mathcal{G}$-observability.

methodology is also suitable for reconfigurable systems. Finally, note that our method can be used in complement to the one of [dC00b] and [AFF02] whenever the sub-specifications do not concern the whole system (i.e for each sub-specification, our method can be used to compute the controllers on the concerned sub-machines, without having to build the corresponding global LTS). However, for some control problems, it may happen that the specification that has to be ensured is more related to the notion of states rather than to the notion of trajectories of the system (the mutual exclusion problem for example). For this class of problems, one of the main issues is the State Avoidance Control Problem or dually the Invariance Control Problem. If one wants to use a language-based approach to encode this problem, then the obtained specification may be of the size of the global system itself which renders the use of the above works intractable in the sense that the computation of the specification requires the computation of the whole system. Hence, the previous method is not suitable for this kind of control problems (see [C25] for a methodology totally devoted to the state avoidance control problem).

In the next section, we relax the synchronous hypothesis and show how to solve the Supervisory Control Problem whenever the communication between sub-systems takes times.

3.3 Control of Distributed Systems

In this section, we consider that the system to be controlled is composed of n (finite) subsystems that communicate through reliable unbounded FIFO channels (meaning that the delay between the transmission and the reception of a message between subsystems is *a priori* unbounded). These subsystems are modeled by *communicating finite state machines* [BZ83] (CFSM for short), a classical model for distributed systems like communication protocols [BZ83, PP91, LGJJ06] and web services [MW07]. Following the architecture described in Figure 3.9, we assume that each subsystem is controlled by a *local* controller which only observes the actions fired by its subsystem and communicates with it with zero delays. The control decision is based on the knowledge each local controller has about the current state of the whole system. Controllers communicate with each other by adding some extra information (some timestamps and their state estimates) to the messages normally exchanged by the subsystems. These communications allow them to refine their knowledge, so that control decisions may be more permissive.
In this section, we focus on the *state avoidance control problem* that consists in preventing the system from reaching some bad states. To solve this control problem, we first compute offline (i.e. before the system execution), the set of states that lead to bad states by only taking uncontrollable transitions. We then compute online (i.e. during the execution of the controlled system) state estimates for each controller so that they can take a better control decision. Since the (co-)reachability problem is undecidable in our settings, we rely on the abstract interpretation techniques of [LGJJ06] to ensure the termination of the computations of our algorithms by overapproximating the possible FIFO channel contents (and hence the state estimates) by regular languages.

Related Works Over the past years a considerable research effort has been done in decentralized supervisory control [RW92, YL00, Ric00, JK00] that allows to synthesize individ-

ual controllers that have a partial observation of the system's moves and can communicate with each other [Ric00, BL00, LRL07]. The pioneer work of Pnueli and Rosner [PR90] shows that the synthesis of distributed systems is in general undecidable. In [GSZ09], Gastin *et al.* study the decidability of LTL synthesis depending on the architecture of the distributed system. However, in these works the authors consider a synchronous architecture between the controllers. In [Tri04], Tripakis studies the decidability of the existence of controllers such that a set of response properties is satisfied in a decentralized framework with communication delays between the controllers. He shows that the problem is undecidable when there is no communication or when the communication delays are unbounded. In [Ira09], Irasihi proves the decidability of a decentralized control problem of discrete event systems with k-bounded-delay communication. In [BBG$^+$10], Bensalem *et al.* propose a knowledge-based algorithm for distributed control: each subsystem is controlled according to a (local) knowledge of the property to ensure. When local knowledge is not sufficient, synchronizations are added until a decision can be taken (the reachability problem is decidable in their model). Unlike them, the reachability problem is undecidable in our model, the state estimates are a form of knowledge that does not depend on the property to ensure, and we never add synchronizations.

The control of concurrent systems (see the previous section) is closely related to our framework [JK00, KvS05, LK02],[J14]. However, in this setting, the system is composed of several subsystems that communicate with zero delay (and similarly for the controllers) whereas in our framework, the subsystems and the controllers communicate asynchronously and we thus have to take into account the *a priori unbounded* communication delays to perform the computation of the controllers.

Our problem differs from the synthesis problem (see e.g. [Mas91, Gen05]) which asks to synthesize a communication protocol and to distribute the actions of a specification depending on the subsystem where they must be executed, and to synchronize them in such a way that the resulting distributed system is equivalent to the given global specification.

In [Dar05], Darondeau synthesizes distributed controllers for distributed system communicating by bounded channels. He states a sufficient condition allowing to decide if a controller can be implemented by a particular class of Petri nets that can be further translated into communicating automata. Some other works deal with the computation of a state estimate of a centralized system with distributed controllers. For example, in [XK09],

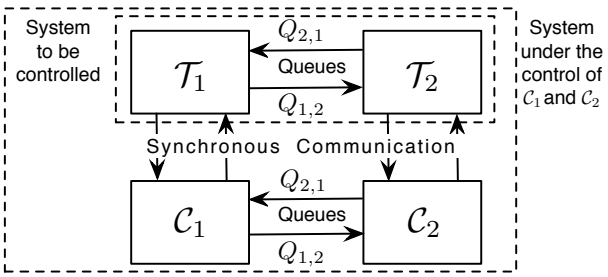


Figure 3.9: Control architecture of a distributed system.

Xu and Kumar propose a distributed algorithm which computes an estimate of the current state of a system. Local estimators maintain and update local state estimates from their own observation of the system and information received from the other estimators. In their framework, the local estimators communicate between each others through reliable FIFO channels asynchronously, whereas the system is monolithic, and therefore these FIFO channels are not included into the global states of the system. Moreover, as we consider concurrent systems, we also have to take account the communication delay between subsystems to compute the state-estimates as well as the control policies. Finally, compared with [XK09], we have chosen to exchange information between controllers using existing communication channels between subsystems. This completely changes the computation of the state-estimates. Note also that the global state estimate problem of a distributed system is related to the problems of (Mazurkiewicz) *trace model checking* and *global predicate detection*; this later aims to check if there exists a possible global configuration of the system that satisfies a given global predicate ϕ. A lot of related works, consider an offline approach where the execution, given as a Mazurkiewicz trace [Maz86] is provided from the beginning (see e.g. [GMM06, KMMB07] for a review and efficient methods). Online global predicate detection has been studied, e.g. in [JTGVR94, SG02]. The proposed solution involves a central monitor which receives on the fly the execution trace. Note that one of the main issues in these problems is to get a precise estimation on the sequences of events in the distributed execution. Therefore, standard techniques based e.g. on vector clocks [Fid88, Mat89] are used to generate a partial ordering of events; and so does also our method. However, compared to the above mentioned works, our problem is particular for several reasons. First, the information must be received by all local controllers since no central monitor is present; second FIFO queues are part of the global states; finally these controllers must take proactive measures to prevent the system from taking an unsafe action.

3.3.1 Model of the system

We model distributed systems by *communicating finite state machines* (CFSMs) [BZ83] with reliable unbounded FIFO channels (also called *queues* below). CFSMs with unbounded channels are very useful to model and verify communication protocols at an abstract level, since we can reason about them without considering the actual size of the queues, which depends on the implementation of the protocol.

Definition 3.8 (Communicating Finite State Machines) *A* CFSM $\mathcal{T}$ *is defined by a 6-tuple* $\langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$*, where* (i) L *is a finite set of locations,* (ii) $\ell_0 \in L$ *is the initial location,* (iii) Q *is a finite set of queues,* (iv) M *is a finite set of messages,* (v) $\Sigma \subseteq Q \times \{!, ?\} \times M$ *is a finite set of actions, which are either an* output $i!m$ *to specify that the message* $m \in M$ *is written on the queue* $i \in Q$ *or an* input $i?m$ *to specify that the message* $m \in M$ *is read on the queue* $i \in Q$*, and* (vi) $\Delta \subseteq L \times \Sigma \times L$ *is a finite set of transitions.*

An *output transition* $\langle \ell, i!m, \ell' \rangle$ indicates that, when the system moves from the location ℓ to ℓ', a message m must be concatenated to the end of the queue i. An *input transition* $\langle \ell, i?m, \ell' \rangle$ indicates that, when the system moves from ℓ to ℓ', the message m must be present at the beginning of the queue i and must be removed from this queue. To simplify

the presentation of our method, this model has no internal actions (i.e. events that are local to a subsystem and that are neither inputs nor outputs) and we assume that $\mathcal{T}$ is deterministic i.e., $\forall \ell \in L, \forall \sigma \in \Sigma : |\{\ell' \in L | \langle \ell, \sigma, \ell' \rangle \in \Delta\}| \leq 1$. Those restrictions are not mandatory and our implementation [McS10] accepts CFSMs with internal actions and non-deterministic ones. For $\sigma \in \Sigma$, the set of transitions of $\mathcal{T}$ labeled by σ is denoted by $\mathrm{Trans}(\sigma)$. The occurrence of a transition is called an *event* and given an event e, δ_e denotes the corresponding transition.

Semantics The semantics of a CFSM is defined as follows: A *global state* in this model is given by the local state of each subsystem together with the content of each queue. Therefore, since no bound is given neither in the transmission delay, nor on the length of the queues, the global state space is a priori infinite. A *global state* of a CFSM $\mathcal{T}$ is a tuple $\langle \ell, w_1, ..., w_{|Q|} \rangle \in \mathcal{D} = L \times (M^*)^{|Q|}$ where ℓ is the current location of $\mathcal{T}$ and $w_1, ..., w_{|Q|}$ are finite words on M^* which give the contents of the queues in Q.

Definition 3.9 (Semantics of a CFSM) *The semantics of a CFSM* $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$ *is given by an LTS* $[\![\mathcal{T}]\!] = \langle \mathcal{D}, \vec{x}_0, \Sigma, \rightarrow \rangle$, *where* (i) $\mathcal{D} \triangleq L \times (M^*)^{|Q|}$ *is the set of states,* (ii) $\vec{x}_0 \triangleq \langle \ell_0, \epsilon, \ldots, \epsilon \rangle$ *is the initial state,* (iii) Σ *is the set of actions, and* (iv) $\rightarrow \triangleq \bigcup_{\delta \in \Delta} \xrightarrow{\delta} \subseteq \mathcal{D} \times \Sigma \times \mathcal{D}$ *is the transition relation where* $\xrightarrow{\delta}$ *is defined as follows:*

$$\frac{\delta = \langle \ell, i!m, \ell' \rangle \in \Delta \quad w_i' = w_i \cdot m}{\langle \ell, w_1, \ldots, w_i, \ldots, w_{|Q|} \rangle \xrightarrow{\delta} \langle \ell', w_1, \ldots, w_i', \ldots, w_{|Q|} \rangle}$$

$$\frac{\delta = \langle \ell, i?m, \ell' \rangle \in \Delta \quad w_i = m \cdot w_i'}{\langle \ell, w_1, \ldots, w_i, \ldots, w_{|Q|} \rangle \xrightarrow{\delta} \langle \ell', w_1, \ldots, w_i', \ldots, w_{|Q|} \rangle}$$

To simplify the notations, we often denote transition $\vec{x} \xrightarrow{\delta_e} \vec{x}'$ by $\vec{x} \xrightarrow{e} \vec{x}'$. An *execution* of $\mathcal{T}$ is a sequence $\vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} \vec{x}_m$ where $\vec{x}_0 = \langle \ell_0, \varepsilon, \cdots, \varepsilon \rangle$ is the only initial state and $\vec{x}_i \xrightarrow{e_{i+1}} \vec{x}_{i+1} \in \rightarrow \forall i \in [0, m-1]$. Given a sequence of actions $\overline{\sigma} = \sigma_1 \cdots \sigma_m \in \Sigma^*$ and two states $x, x' \in X$, $x \xrightarrow{\overline{\sigma}} x'$ denotes that the state x' is reachable from x by executing $\overline{\sigma}$. Given a set of states $Y \subseteq X$ and $\Delta' \subseteq \Delta$, $\mathsf{Reach}^{\mathcal{T}}_{\Delta'}(Y)$ corresponds to the set of states that are reachable in $[\![\mathcal{T}]\!]$ from Y by only triggering transitions of $\Delta' \subseteq \Delta$ in $\mathcal{T}$, whereas $\mathrm{Coreach}^{\mathcal{T}}_{\Delta'}(Y)$ denotes the set of states from which Y is reachable by only triggering transitions of Δ':

$$\mathsf{Reach}^{\mathcal{T}}_{\Delta'}(E) \triangleq \bigcup_{n \geq 0} \mathrm{Post}^{\mathcal{T}^n}_{\Delta'}(E) \tag{3.8}$$

$$\mathrm{Coreach}^{\mathcal{T}}_{\Delta'}(E) \triangleq \bigcup_{n \geq 0} \mathrm{Pre}^{\mathcal{T}^n}_{\Delta'}(E) \tag{3.9}$$

where $(\mathrm{Post}^{\mathcal{T}}_{\Delta'}(E))^n$ and $(\mathrm{Pre}^{\mathcal{T}}_{\Delta'}(E))^n$ are the n^{th} functional power of $\mathrm{Post}^{\mathcal{T}}_{\Delta'}(E) \triangleq \{\vec{x}' \in \mathcal{D} | \exists \vec{x} \in E, \exists \delta \in \Delta' : \vec{x} \xrightarrow{\delta} \vec{x}'\}$ and $\mathrm{Pre}^{\mathcal{T}}_{\Delta'}(E) \triangleq \{\vec{x}' \in \mathcal{D} | \exists \vec{x} \in E, \exists \delta \in \Delta' : \vec{x}' \xrightarrow{\delta} \vec{x}\}$. Although there is no general algorithm that can exactly compute the (co)reachability set [BZ83], there exists a technique that allows us to compute an over-approximation of this set (see section 3.3.4.2).

Product of CFSM A distributed system $\mathcal{T}$ is generally composed of several subsystems $\mathcal{T}_i$ ($\forall i \in [1..n]$). In our case, this global system $\mathcal{T}$ is defined by a CFSM resulting from the product of the n subsystems $\mathcal{T}_i$, also modeled by CFSMs. This can be defined through the product of two subsystems.

Definition 3.10 (Product) *Given two CFSMs $\mathcal{T}_i = \langle L_i, \ell_{0,i}, Q_i, M_i, \Sigma_i, \Delta_i \rangle$, their product, denoted by $\mathcal{T}_1 || \mathcal{T}_2$, is defined by a CFSM $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$, where (i) $L \triangleq L_1 \times L_2$, (ii) $\ell_0 \triangleq (\ell_{0,1}, \ell_{0,2})$, (iii) $Q \triangleq Q_1 \cup Q_2$, (iv) $M \triangleq M_1 \cup M_2$, (v) $\Sigma \triangleq \Sigma_1 \cup \Sigma_2$, and (vi) $\Delta \triangleq \{\langle\langle \ell_1, \ell_2 \rangle, \sigma_1, \langle \ell'_1, \ell_2 \rangle\rangle | (\langle \ell_1, \sigma_1, \ell'_1 \rangle \in \Delta_1) \wedge (\ell_2 \in L_2)\} \cup \{\langle\langle \ell_1, \ell_2 \rangle, \sigma_2, \langle \ell_1, \ell'_2 \rangle\rangle | (\langle \ell_2, \sigma_2, \ell'_2 \rangle \in \Delta_2) \wedge (\ell_1 \in L_1)\}$.*

This operation is associative and commutative up to state renaming.

Definition 3.11 (Distributed system) *A distributed system $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$ is defined by the product of n CFSMs $\mathcal{T}_i = \langle L_i, \ell_{0,i}, N_i, M, \Sigma_i, \Delta_i \rangle$ ($\forall i \in [1..n]$) acting in parallel and exchanging information through FIFO channels.*

Note that a distributed system is also modeled by a CFSM, since the product of several CFSMs is a CFSM. To avoid the confusion between the model of one subsystem and the model of the whole system, in the sequel, a CFSM $\mathcal{T}_i$ always denotes the model of a single process, and a CFSM $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$ always denotes the distributed system $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$.

Communication Architecture We consider an architecture for the system $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$ (See Def. 3.11 with *point-to-point* communication i.e., any subsystem $\mathcal{T}_i$ can send messages to any other subsystem $\mathcal{T}_j$ through a queue[8] $Q_{i,j}$. Thus, only $\mathcal{T}_i$ can write a message m on $Q_{i,j}$ (denoted by $Q_{i,j}!m$) and only $\mathcal{T}_j$ can read m on this queue (denoted by $Q_{i,j}?m$). Moreover, we suppose that the queues are unbounded, that the message transfers between the subsystems are reliable and may suffer from arbitrary non-zero delays, and that no *global clock* or *perfectly synchronized local clocks* are available. With this architecture[9], the set Q_i of $\mathcal{T}_i$ ($\forall i \in [1..n]$) can be rewritten as $Q_i = \{Q_{i,j}, Q_{j,i} \,|\, (1 \leq j \leq n) \wedge (j \neq i)\}$ and $\forall j \neq i \in [1..n]$, $\Sigma_i \cap \Sigma_j = \emptyset$. Let $\delta_i = \langle \ell_i, \sigma_i, \ell'_i \rangle \in \Delta_i$ be a transition of $\mathcal{T}_i$, $\mathsf{global}(\delta_i) \triangleq \{\langle\langle \ell_1, \ldots, \ell_{i-1}, \ell_i, \ell_{i+1}, \ldots, \ell_n \rangle, \sigma_i, \langle \ell_1, \ldots, \ell_{i-1}, \ell'_i, \ell_{i+1}, \ldots, \ell_n \rangle\rangle \in \Delta \,|\forall j \neq i \in [1..n] : \ell_j \in L_j\}$ is the set of transitions of Δ that can be built from δ_i in $\mathcal{T}$. We extend this definition to sets of transitions $D \subseteq \Delta_i$ of the subsystem $\mathcal{T}_i$: $\mathsf{global}(D) \triangleq \bigcup_{\delta_i \in D} \mathsf{global}(\delta_i)$. We abuse notation and write $\Delta \setminus \Delta_i$ instead of $\Delta \setminus \mathsf{global}(\Delta_i)$ to denote the set of transitions of Δ that are not built from Δ_i.

3.3.2 Framework and State Avoidance Control Problem

In the sequel, we are interested in the state avoidance control problem which consists in preventing the system from reaching some undesirable states.

[8] To simplify the presentation of our method, we assume that there is one queue from $\mathcal{T}_i$ to $\mathcal{T}_j$. But, our implementation is more permissive and zero, one or more queues can exist from $\mathcal{T}_i$ to $\mathcal{T}_j$.

[9] In our examples, we do not mention queue $Q_{i,j}$ when there is no message sent from $\mathcal{T}_i$ to $\mathcal{T}_j$.

3.3.2.1 Control Architecture

The distributed system $\mathcal{T}$ is composed of n subsystems $\mathcal{T}_i$ ($\forall i \in [1..n]$) and we want to associate a local controller $\mathcal{C}_i$ with each subsystem $\mathcal{T}_i$ in order to satisfy the control requirements. Each controller $\mathcal{C}_i$ interacts with $\mathcal{T}_i$ in a feedback manner: $\mathcal{C}_i$ observes the last action fired by $\mathcal{T}_i$ and computes, from this observation and some information received from the other controllers (corresponding to some state estimates), a set of actions that $\mathcal{T}_i$ cannot fire in order to ensure the desired properties on the global system. Following the Ramadge & Wonham's theory [RW89], the set of actions Σ_i of $\mathcal{T}_i$ is partitioned into the set of controllable actions $\Sigma_{i,c}$, that can be forbidden by $\mathcal{C}_i$, and the set of uncontrollable actions $\Sigma_{i,uc}$, that cannot be forbidden by $\mathcal{C}_i$. The subsets $\Sigma_{1,c}, \ldots, \Sigma_{n,c}$ are disjoint, because $\Sigma_i \cap \Sigma_j = \emptyset, \forall i \neq j \in [1..n]$. In this section and in our implementation [McS10], inputs are uncontrollable and outputs are controllable, a classical assumption for reactive systems. Our algorithm however does not depend on this particular partition of the actions, since one of its parameters is the set of uncontrollable actions. The set of actions, that can be controlled by at least one controller, is denoted by Σ_c and is defined by $\Sigma_c \triangleq \bigcup_{i=1}^{n} \Sigma_{i,c}$; We also define $\Sigma_{uc} \triangleq \Sigma \setminus \Sigma_c = \bigcup_{i=1}^{n} \Sigma_{i,uc}$. This partition also induces a partition on the set of transitions Δ_i into the sets $\Delta_{i,c}$ and $\Delta_{i,uc}$. The set of transitions Δ is similarly partitioned into the sets Δ_c and Δ_{uc}.

3.3.2.2 Distributed Controller and Controlled Execution

The control decision depends on the current state of the global system $\mathcal{T}$ (i.e. state-feedback control). Unfortunately, a local controller does not generally know the current global state, due to its partial observation of the system. So, it must define its control policy from a *state estimate* corresponding to its evaluation of the states in which the system $\mathcal{T}$ can be. It is formally defined as follows:

Definition 3.12 (Local Controller) *A* local controller *is a function* $\mathcal{C}_i : 2^{\mathcal{D}} \rightarrow 2^{\Sigma_{i,c}}$ *which defines, for each estimate* $E \in 2^{\mathcal{D}}$ *of the current state of* $\mathcal{T}$*, the set of controllable actions that* $\mathcal{T}_i$ *may not execute.*

This definition of a local controller does not explain how each local controller can compute a state estimate. In section 3.3.3, we define an algorithm that allows $\mathcal{C}_i$ to compute this state estimate during the execution of this system. Note that besides the precision of the state estimate, one important property that should be satisfied by the state estimate E is that the actual current state of the system belongs to E.
Based on Definition 3.12, a *distributed controller* is defined by:

Definition 3.13 (Distributed Controller) *A* distributed controller $\mathcal{C}_{\mathsf{di}}$ *is defined by a tuple* $\mathcal{C}_{\mathsf{di}} \triangleq \langle \mathcal{C}_i \rangle_{i=1}^{n}$ *where each* $\mathcal{C}_i,\ i \in [1..n]$ *is a local controller.*

A *controlled execution* is an execution that may occur in $\mathcal{T}$ under the control of $\mathcal{C}_{\mathsf{di}}$.

Definition 3.14 (Controlled Execution) *Given a distributed controller* $\mathcal{C}_{\mathsf{di}} = \langle \mathcal{C}_i \rangle_{i=1}^{n}$, $s = \vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} \vec{x}_m$ *is a* controlled execution *of* $\mathcal{T}$ *under the control of* $\mathcal{C}_{\mathsf{di}}$ *if* $\forall k \in [1, m]$, *whenever* $\delta_{e_k} \in \Delta_i$ *and the state estimate* $\vec{x}_{k-1}$ *is* E*, then* $\sigma_{e_k} \notin \mathcal{C}_i(E)$.

Note that with this definition, the language of the controlled system is controllable with respect to the language of the original system. It is basically due to the fact that each local controller is only able to disable the controllable actions that can occur in its corresponding subsystem.

3.3.2.3 Definition of the Control Problem

Control synthesis aims at restricting the behavior of a system to satisfy a property. The properties we consider are invariance properties, defined by a subset $Good \subseteq \mathcal{D}$ of states, in which any execution of the transition system should be confined. Alternatively, it can be viewed as a state avoidance property where $Bad = \mathcal{D} \setminus Good$ defines a set of states that no execution should reach. Notice that the specification Bad can involve the contents of the FIFO channels (we remember that $\mathcal{D} = L \times (M^*)^{|Q|}$). We define the problem as follows:

Problem 3.2 (Distributed State Avoidance Control Problem) *Given a set $Bad \subseteq \mathcal{D}$ of forbidden states, the* distributed state avoidance control problem *(the* distributed problem *for short) consists in synthesizing a distributed controller $\mathcal{C}_{\mathsf{di}} = \langle \mathcal{C}_i \rangle_{i=1}^{n}$ such that each controlled execution of the system $\mathcal{T}$ under the control of $\mathcal{C}_{\mathsf{di}}$ avoids Bad.*

Proposition 3.2 *Given a distributed systems $\mathcal{T}$, a distributed controller $\mathcal{C}_{\mathsf{di}}$ and a set of forbidden states $Bad \subseteq \mathcal{D}$, it is undecidable to know whether $\mathcal{C}_{\mathsf{di}}$ solves Problem 3.2. Moreover, deciding the existence of a non-trivial controller $\mathcal{C}_{\mathsf{di}}$ solving Problem 3.2 is undecidable.*

Intuitively, this result is a consequence of the undecidability of the (co-)reachability problem in the CFSM model [BZ83].

Remark 3.3 (Trivial solution and the non-blocking problem) *The definition of Problem 3.2 does not tackle the non-blocking problem (i.e. by imposing that at every time at least one transition of one of the sub-systems is allowed). Therefore, there exists a trivial solution of this problem, which consists in disabling all output transitions so that nothing happens in the controlled system. However, our aim is to find, as often as possible, solutions that are correct and permissive enough to be of practical value. Since the principle of safe control is to allow a transition only when the local controller is sure that this transition cannot lead to a bad state, permissiveness directly depends on the knowledge local controllers have about the global system.*

3.3.3 State Estimates of Distributed Systems

In this section, we present an algorithm that computes estimates of the current state of a distributed system. The result of this algorithm is used, in section 3.3.4, by our control algorithm which synthesizes distributed controllers for the distributed problem. We first recall the notion of *vector clocks* [Lam78], a standard concept that we use to compute state estimates.

3.3.3.1 Vector Clocks

To allow the local controllers to have a better understanding of an execution of the distributed system, it is important to determine the causal and temporal relationship between the events that occur during this execution : events emitted by a same subsystem are ordered. When concurrent subsystems communicate, additional ordering information can be obtained, and the communication scheme can be used to obtain a partial order on the events of the system. In practice, vectors of logical clocks, called *Vector clocks* [Lam78], can be used to time-stamp the events of a distributed system. The order of the vector clocks induces the order of the corresponding events. Vector clocks are formally defined as follows:

Definition 3.15 (Vector Clocks) *Let $\langle D, \sqsubseteq \rangle$ be a partially ordered set, a* vector clock mapping *of width n is a function $V : D \mapsto \mathbb{N}^n$ such that $\forall d_1, d_2 \in D : (d_1 \sqsubseteq d_2) \Leftrightarrow (V(d_1) \leq V(d_2))$.*

In general, for a distributed system composed of n subsystems, the partial order on events is represented by a vector clock mapping of width n. The method for computing this vector clock mapping depends on the communication scheme of the distributed system. For CFSMs, it can be computed by the Mattern's algorithm [Mat89], which is based on the causal and thus temporal relationship between the sending and reception of any message transferred through any FIFO channel. This information is then used to determine a partial order, called *causality (or happened-before) relation* $\prec_c$, on the events of the distributed system. This relation is the smallest transitive relation satisfying the following conditions: (i) if the events $e_i \neq e_j$ occur in the same subsystem $\mathcal{T}_i$ and if e_i comes before e_j in the execution, then $e_i \prec_c e_j$, and (ii) if e_i is an output event occurring in $\mathcal{T}_i$ and if e_j is the corresponding input event occurring in $\mathcal{T}_j$, then $e_i \prec_c e_j$. In the sequel, when $e_i \prec_c e_j$, we say that e_j *causally depends* on e_i (or e_i *happened-before* e_j).
In Mattern's algorithm [Mat89], each subsystem $\mathcal{T}_i$ ($\forall i \in [1..n]$) maintains a vector clock $V_i \in \mathbb{N}^n$. Each element $V_i[j], \forall j \in [1..n]$ is a counter which represents the knowledge of $\mathcal{T}_i$ regarding T_j and which can roughly be interpreted as follows: $\mathcal{T}_i$ knows that $\mathcal{T}_j$ has executed at least $V_i[j]$ events. Initially, each component of the vector V_i, $\forall i \in [1..n]$ is set to 0. Next, when an event e occurs in $\mathcal{T}_i$, the vector clock V_i is updated as follows: first, $V_i[i]$ is incremented (i.e., $V_i[i] \leftarrow V_i[i] + 1$) to indicate that a new event occurred in $\mathcal{T}_i$ and next two cases are considered:

- if e consists in sending a message m to $\mathcal{T}_j$, the vector clock V_i is attached to m and both information are sent to $\mathcal{T}_j$.
- if e corresponds to the reception of a message m tagged with vector clock V_j, then V_i is set to the component-wise maximum of V_i and V_j. This allows us to take into account the fact that any event, that precedes the sending of m, should also precede the event e.

We now define a lemma related to vector clocks that will be used in the sequel:

Lemma 3.1 Given a sequence $se_1 = \vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_{i-1}} \vec{x}_{i-1} \xrightarrow{e_i} \vec{x}_i \xrightarrow{e_{i+1}} \vec{x}_{i+1} \xrightarrow{e_{i+2}} \ldots \xrightarrow{e_m} \vec{x}_m$ executed by $\mathcal{T}$, if $e_i \not\prec_c e_{i+1}$, then the sequence $se_2 = \vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_{i-1}} \vec{x}_{i-1} \xrightarrow{e_{i+1}} \vec{x}'_i \xrightarrow{e_i} \vec{x}_{i+1} \xrightarrow{e_{i+2}} \ldots \xrightarrow{e_m} \vec{x}_m$ can also occur in $\mathcal{T}$.

This property means that if two consecutive events e_i and e_{i+1} are such that $e_i \not\prec_c e_{i+1}$, then these events can be swapped without modifying the reachability of $\vec{x}_m$.

3.3.3.2 Computation of State Estimates

Each time an event occurs in a sub-system $\mathcal{T}_i$, the local controller $\mathcal{C}_i$ updates its vector clock V_i and its state estimate E_i that should contain the current state of $\mathcal{T}$. Note that E_i must also contain any future state that can be reached from this current state by firing actions that do not belong to $\mathcal{T}_i$ as E_i does not observe them. Our state estimate algorithm proceeds as follows :

- When $\mathcal{T}_i$ sends a message m to $\mathcal{T}_j$, $\mathcal{T}_i$ attaches the vector clock V_i and the state estimate E_i of $\mathcal{C}_i$ to this message. Next, $\mathcal{C}_i$ observes the action fired by $\mathcal{T}_i$, and infers the fired transition. It then uses this information to update its state estimate E_i.
- When $\mathcal{T}_i$ receives a message m from $\mathcal{T}_j$, $\mathcal{C}_i$ observes the action fired by $\mathcal{T}_i$ and the information sent by $\mathcal{T}_j$ i.e., the state estimate E_j and the vector clock V_j of $\mathcal{C}_j$. It computes its new state estimate from these elements.

In both cases, the computation of the new state estimate E_i depends on the computation of reachable states. In this section, we assume that we have an operator that can compute an *approximation* of the reachable states. We explain in section 3.3.4 how to implement this operator.

State Estimate Algorithm Our algorithm, called *SE-algorithm*, computes state estimates of a distributed system. It is composed of three sub-algorithms: (i) the initialEstimate algorithm, which is only used when the system starts its execution, computes, for each controller, its initial state estimate (ii) the outputTransition algorithm computes on-line the new state estimate of $\mathcal{C}_i$ after an output of $\mathcal{T}_i$, and (iii) the inputTransition algorithm computes online the new state estimate of $\mathcal{C}_i$ after an input of $\mathcal{T}_i$.
initialEstimate *Algorithm:* Each component of the vector V_i is set to 0. To take into account that, before the execution of the first action of $\mathcal{T}_i$, the other subsystems $\mathcal{T}_j$ ($\forall j \neq i \in [1..n]$) could perform inputs and outputs, the initial state estimate of $\mathcal{C}_i$ is given by $E_i = \mathsf{Reach}^{\mathcal{T}}_{\Delta\setminus\Delta_i}(\langle \ell_{0,1}, \ldots, \ell_{0,n}, \epsilon, \ldots, \epsilon\rangle)$.
outputTransition *Algorithm:* Let E_i be the current state estimate of $\mathcal{C}_i$. When $\mathcal{T}_i$ fires an output transition $\delta = \langle \ell_1, Q_{i,j}!m, \ell_2\rangle \in \Delta_i$, the following operations are performed to

Algorithm 2: initialEstimate($\mathcal{T}$)

input : $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$.
output: The initial state estimate E_i of the controller $\mathcal{C}_i$ ($\forall i \in [1..n]$).

```
begin
    for i ← 1 to n do  for j ← 1 to n do  V_i[j] ← 0
    for i ← 1 to n do  E_i ← Reach^T_{Δ\Δ_i}(⟨ℓ_{0,1}, ..., ℓ_{0,n}, ϵ, ..., ϵ⟩)
    return (E_1, ..., E_n)
end
```

Algorithm 3: outputTransition$(\mathcal{T}, V_i, E_i, \delta)$

input : $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$, the vector clock V_i of $\mathcal{C}_i$, the current state estimate E_i of $\mathcal{C}_i$, and a transition $\delta = \langle \ell_1, Q_{i,j}!m, \ell_2 \rangle \in \Delta_i$.
output: The state estimate E_i after the output transition δ.

```
1 begin
2     V_i[i] ← V_i[i] + 1
3     T_i tags message m with ⟨E_i, V_i, δ⟩ and writes this tagged message on Q_{i,j}
4     E_i ← Reach^T_{Δ\Δ_i}(Post^T_δ(E_i))
5     return (E_i)
6 end
```

update the state estimate E_i:

- $V_i[i]$ is incremented (i.e., $V_i[i] \leftarrow V_i[i] + 1$) to indicate that a new event has occurred in $\mathcal{T}_i$.

- $\mathcal{T}_i$ tags message m with $\langle E_i, V_i, \delta \rangle$ and writes this information on $Q_{i,j}$. The estimate E_i, tagging m, contains the set of states in which $\mathcal{T}$ can be *before* the execution of δ. The additional information $\langle E_i, V_i, \delta \rangle$ will be used by $\mathcal{T}_j$ to refine its state estimate.

- E_i is updated as follows to contain the current state of $\mathcal{T}$ and any future state that can be reached from this current state by firing actions that do not belong to $\mathcal{T}_i$: $E_i \leftarrow \mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathrm{Post}^{\mathcal{T}}_{\delta}(E_i))$. More precisely, $\mathrm{Post}^{\mathcal{T}}_{\delta}(E_i)$ gives the set of states in which $\mathcal{T}$ can be after the execution of δ. But, after the execution of this transition, $\mathcal{T}_j$ $(\forall j \neq i \in [1..n])$ could read and write on their queues. Therefore, we define the estimate E_i by $\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathrm{Post}^{\mathcal{T}}_{\delta}(E_i))$.

inputTransition *Algorithm:* Let E_i be the current state estimate of $\mathcal{C}_i$. When $\mathcal{T}_i$ fires an input transition $\delta = \langle \ell_1, Q_{j,i}?m, \ell_2 \rangle \in \Delta_i$, it also reads the information $\langle E_j, V_j, \delta' \rangle$ (where E_j is the state estimate of $\mathcal{C}_j$ before the execution of δ' by $\mathcal{T}_j$, V_j is the vector clock of $\mathcal{C}_j$ after the execution of δ' by $\mathcal{T}_j$, and $\delta' = \langle \ell'_1, Q_{j,i}!m, \ell'_2 \rangle \in \Delta_j$ is the output corresponding to δ) tagging m, and the following operations are performed to update E_i:

- we update the state estimate E_j of $\mathcal{C}_j$ (this update is stored in $Temp$) by using the vector clocks to guess the possible behaviors of $\mathcal{T}$ between the execution of the transition δ' and the execution of δ. We consider three cases :

  - if $V_j[i] = V_i[i]$: $Temp \leftarrow \mathrm{Post}^{\mathcal{T}}_{\delta}(\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathrm{Post}^{\mathcal{T}}_{\delta'}(E_j)))$. In this case, thanks to the vector clocks, we know that $\mathcal{T}_i$ has executed no transition between the execution of δ' by $\mathcal{T}_j$ and the execution of δ by $\mathcal{T}_i$. Thus, only transitions in $\Delta \setminus \Delta_i$ could have occurred during this period. We then update E_j as follows. We compute (i) $\mathrm{Post}^{\mathcal{T}}_{\delta'}(E_j)$ to take into account the execution of δ' by $\mathcal{T}_j$, (ii) $\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathrm{Post}^{\mathcal{T}}_{\delta'}(E_j))$ to take into account the transitions that could occur between the execution of δ' and the execution of δ, and (iii) $\mathrm{Post}^{\mathcal{T}}_{\delta}(\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathrm{Post}^{\mathcal{T}}_{\delta'}(E_j)))$ to take into account the execution of δ.

Algorithm 4: inputTransition($\mathcal{T}, V_i, E_i, \delta$)

input : $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$, the vector clock V_i of $\mathcal{C}_i$, the current state estimate E_i of $\mathcal{C}_i$ and a transition $\delta = \langle \ell_1, Q_{j,i}?m, \ell_2 \rangle \in \Delta_i$. Message m is tagged with the triple $\langle E_j, V_j, \delta' \rangle$ where (i) E_j is the state estimate of $\mathcal{C}_j$ before the execution of δ' by $\mathcal{T}_j$, (ii) V_j is the vector clock of $\mathcal{C}_j$ after the execution of δ' by $\mathcal{T}_j$, and (iii) $\delta' = \langle \ell'_1, Q_{j,i}!m, \ell'_2 \rangle \in \Delta_j$ is the output corresponding to δ.

output: The state estimate E_i after the input transition δ.

```
1  begin
2  |  \\ We consider three cases to update E_j
3  |  if V_j[i] = V_i[i] then Temp ← Post^T_δ(Reach^T_{Δ\Δ_i}(Post^T_{δ'}(E_j)))
4  |  else if V_j[j] > V_i[j] then Temp ← Post^T_δ(Reach^T_{Δ\Δ_i}(Reach^T_{Δ\Δ_j}(Post^T_{δ'}(E_j))))
5  |  else Temp ← Post^T_δ(Reach^T_Δ(Post^T_{δ'}(E_j)))
6  |  E_i ← Post^T_δ(E_i) \\ We update E_i
7  |  E_i ← E_i ∩ Temp \\ E_i = update of E_i ∩ update of E_j (i.e., Temp)
8  |  V_i[i] ← V_i[i] + 1
9  |  for k ← 1 to n do V_i[k] ← max(V_i[k], V_j[k])
10 end
```

  - else if $V_j[j] > V_i[j]$: $Temp \leftarrow \mathrm{Post}^{\mathcal{T}}_{\delta}(\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_j}(\mathrm{Post}^{\mathcal{T}}_{\delta'}(E_j))))$. Indeed, in this case, we can prove (see Theorem 3.6) that if we reorder the transitions executed between the occurrence of δ' and the occurrence of δ in order to execute the transitions of Δ_i before the ones of Δ_j, we obtain a correct update of E_i. Intuitively, this reordering is possible, because there is no causal relation between the events of $\mathcal{T}_i$ and the events of $\mathcal{T}_j$, that have occurred between δ' and δ. So, in this reordered sequence, we know that, after the execution of δ, only transitions in $\Delta \setminus \Delta_j$ could occur followed by transitions in $\Delta \setminus \Delta_i$.
  - else $Temp \leftarrow \mathrm{Post}^{\mathcal{T}}_{\delta}(\mathsf{Reach}^{\mathcal{T}}_{\Delta}(\mathrm{Post}^{\mathcal{T}}_{\delta'}(E_j)))$. Indeed, in this case, the vector clocks do not allow us to deduce information regarding the behavior of $\mathcal{T}$ between the execution of δ' and the execution of δ. Therefore, to have a correct state estimate, we update E_j by taking into account all the possible behaviors of $\mathcal{T}$ between the execution of δ' and the execution of δ.

- E_i is updated to take into account the execution of δ: $E_i \leftarrow \mathrm{Post}^{\mathcal{T}}_{\delta}(E_i)$.
- and we intersect $Temp$ and E_i to obtain a better state estimate: $E_i \leftarrow E_i \cap Temp$.
- vector clock V_i is incremented to take into account the execution of δ and subsequently is set to the component-wise maximum of V_i and V_j. This last operation allows us to take into account the fact that any event that precedes the sending of m should also precede the occurrence of δ.

3.3.3.3 Properties

State estimate algorithms should have two important properties: soundness and completeness. Completeness means that the current state of the global system is always included in

the state estimates computed by each controller. Soundness means that all states included in the state estimate of $\mathcal{C}_i$ ($\forall i \in [1..n]$) can be reached by one of the sequences of actions that are compatible with the local observation of $\mathcal{T}_i$.

We first introduce some additional notations and a lemma used in the proof of Theorem 3.6. Let $s = \vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \dots \xrightarrow{e_m} \vec{x}_m$ be an execution of $\mathcal{T}$. When an event e_k is executed in the sequence s, the state estimate of *each* controller $\mathcal{C}_i$ is denoted by E_i^k. This state estimate is defined in the following way: if e_k has not been executed by $\mathcal{T}_i$, then $E_i^k \triangleq E_i^{k-1}$. Otherwise, E_i^k is computed by $\mathcal{C}_i$ according to Algorithm 3 or 4.

Theorem 3.6 *The SE-algorithm is complete if the* Reach *operator computes an overapproximation of the reachable states. In other words, the SE-algorithm satisfies the following property: for any execution* $\vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \dots \xrightarrow{e_m} \vec{x}_m$ *of* $\mathcal{T}$, $\vec{x}_m \in \bigcap_{i=1}^{n} E_i^m$.

Theorem 3.7 *The SE-algorithm is sound if the* Reach *operator computes an underapproximation of the reachable states. In other words, the SE-algorithm satisfies the following property: for any execution* $\vec{x}_0 \xrightarrow{e_1} \vec{x}_1 \xrightarrow{e_2} \dots \xrightarrow{e_m} \vec{x}_m$ *of* $\mathcal{T}$, $E_i \subseteq \{x' \in X | \exists \overline{\sigma} \in P_{\Sigma_i}^{-1}(P_{\Sigma_i}(\sigma_{e_1}.\sigma_{e_2} \dots \sigma_{e_m})) : \vec{x}_0 \xrightarrow{\overline{\sigma}} x'\}$ ($\forall i \leq n$) *where* $\forall k \in [1, m]$, σ_{e_k} *is the action that labels the transition corresponding to* e_k.

If we compute an underapproximation of the reachable states, our state estimate algorithm is sound but not complete. If we compute an overapproximation of the reachable states, our state estimate algorithm is complete but not sound. Since we only need completeness to solve the control problem, we define in section 3.3.4 an effective algorithm for the distributed problem by computing overapproximations of the reachable states.

3.3.4 Actual Computation by Means of Abstract Interpretation of Distributed Controllers for the Distributed Problem

In this section, we first define a semi-algorithm for the distributed problem which uses the SE-algorithm as sub-algorithm. Next, we explain how to extend it by using abstract interpretation techniques to obtain an effective algorithm.

3.3.4.1 Semi-Algorithm for the Distributed Problem

Our algorithm, which synthesizes a distributed controller $\mathcal{C}_{\mathsf{di}}$ for the distributed problem, is composed of two parts:

- **Offline part:** We compute the set $I(Bad)$ of states of the global system $\mathcal{T}$ that may lead to Bad by a sequence of uncontrollable transitions. Next, we compute, for each local controller $\mathcal{C}_i$, a control function $\mathcal{F}_i$ which gives, for each action σ of $\mathcal{T}_i$, the set of states of $\mathcal{T}$ that may lead to $I(Bad)$ by a transition labeled by σ. This information is used by $\mathcal{C}_i$, in the online part, to define its control policy.
- **Online part:** During the execution of $\mathcal{T}$, each local controller $\mathcal{C}_i$ uses the SE-algorithm to obtain its own state estimate E_i and computes from this information the actions to be forbidden.

These two parts are formalized as follows.

Offline Part The set $I(Bad)$ of states of $\mathcal{T}$ leading uncontrollably to Bad is given by $\mathrm{Coreach}^{\mathcal{T}}_{\Delta_{uc}}(Bad)$ which, as a reminder, is defined by $\mathrm{Coreach}^{\mathcal{T}}_{\Delta_{uc}}(Bad) = \bigcup_{n\geq 0}(\mathrm{Pre}^{\mathcal{T}}_{\Delta_{uc}})^n(Bad)$ (see (3.9)). Alternatively, it is defined as the least fixpoint of the function $\lambda B.Bad \cup \mathrm{Pre}^{\mathcal{T}}_{\Delta_{uc}}(B)$. Since this function is continuous as a composition of continuous functions, the Knaster-Tarski and Kleene's theorems [Tar55, Mar97] ensure that the least fixpoint exists, so $I(Bad) = \mathrm{Coreach}^{\mathcal{T}}_{\Delta_{uc}}(Bad)$.

Next, we define, for each local controller $\mathcal{C}_i$, the control function $\mathcal{F}_i : \Sigma_i \times 2^X \to 2^X$, which gives, for each action $\sigma \in \Sigma_i$ and set $B \subseteq X$ of states to be forbidden, the set $\mathcal{F}_i(\sigma, B)$ of global states in which the action σ must be forbidden. This set corresponds, more precisely, to the greatest set $\mathcal{O}$ of states of $\mathcal{T}$ such that, for each state $\vec{x} \in \mathcal{O}$, there exists a transition labeled by σ leading to B from $\vec{x}$:

$$\mathcal{F}_i(\sigma, B) \triangleq \begin{cases} \mathrm{Pre}^{\mathcal{T}}_{\mathrm{Trans}(\sigma)}(B) & \text{if } \sigma \in \Sigma_{i,c} \\ \emptyset & \text{otherwise} \end{cases} \tag{3.10}$$

We compute, for each action $\sigma \in \Sigma_i$, the set $\mathcal{F}_i(\sigma, I(Bad))\ \forall i \in [1..n]$. This information is used, during the execution of $\mathcal{T}$, by the local controller $\mathcal{C}_i$ to compute the actions to be forbidden.

Online Part The local controller $\mathcal{C}_i$ is formally defined, for each state estimate $E \in 2^X$, by:

$$\mathcal{C}_i(E) \triangleq \{\sigma \in \Sigma_i \mid \mathcal{F}_i(\sigma, I(Bad)) \cap E \neq \emptyset\} \tag{3.11}$$

Informally, if E is the state estimate of $\mathcal{C}_i$, it forbids an action $\sigma \in \Sigma_i$ if and only if there exists a state $\vec{x} \in E$ in which the action σ must be forbidden in order to prevent the system $\mathcal{T}$ from reaching $I(Bad)$.

During the execution of the system, when the subsystem $\mathcal{T}_i$ ($\forall i \in [1..n]$) executes a transition $\delta = \langle \ell_i, \sigma, \ell_i' \rangle$, the local controller $\mathcal{C}_i$ receives the following information:

- if $\sigma = Q_{j,i}?m$ (with $j \neq i \in [1..n]$), it receives σ, and the triple $\langle E_j, V_j, \delta' \rangle$ tagging m.
- if $\sigma = Q_{i,j}!m$ (with $j \neq i \in [1..n]$), it receives σ.

In both cases, since $\mathcal{C}_i$ knows that $\mathcal{T}_i$ was in the location ℓ_i before triggering σ, this controller can infer the fired transition. $\mathcal{C}_i$ then uses the SE-algorithm with this information to update its state estimate E_i and computes, from this estimate, the set $\mathcal{C}_i(E_i)$ of actions that $\mathcal{T}_i$ cannot execute.

The following theorem proves that this algorithm synthesizes correct controllers for the distributed problem.

Theorem 3.8 *Given a set of forbidden states $Bad \subseteq X$, our distributed controller $\mathcal{C}_{\mathsf{di}} = \langle \mathcal{C}_i \rangle_{i=1}^n$ solves the distributed problem if $\vec{x}_0 \notin I(Bad)$.*

Example 3.5 We consider the sequence of actions of our running example of Figure 3.10. The set Bad is given by the set of global states where the location of

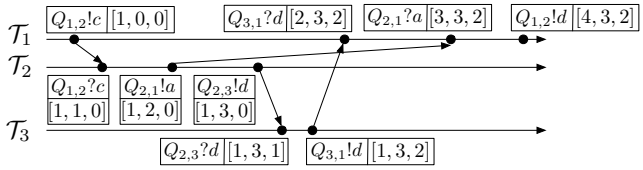


Figure 3.10: An execution of the running example.

$\mathcal{T}_1$ is A_{er}. Thus, $I(Bad) = Bad \cup \{\langle \ell_1, \ell_2, \ell_3, w_{1,2}, w_{2,1}, w_{2,3}, w_{3,1}\rangle | (\ell_1 = A_0) \wedge (w_{2,1} = a.M^*)\}$. At the beginning of the execution of $\mathcal{T}$, the state estimates of the subsystems are $E_1 = \{\langle A_0, B_0, D_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle\}$, $E_2 = \{\langle A_0, B_0, D_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_0, D_0, c, \epsilon, \epsilon, \epsilon\rangle\}$, and $E_3 = \{\langle A_0, B_0, D_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_0, D_0, c, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_1, D_0, \epsilon, b^*, \epsilon, \epsilon\rangle, \langle A_1, B_2, D_0, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle, \langle A_1, B_3, D_0, \epsilon, b^*(a+\epsilon), d, \epsilon\rangle\}$. After the first transition $\langle A_0, Q_{1,2}!c, A_1\rangle$, the state estimate of the controller $\mathcal{C}_1$ is not really precise, because a lot of things may happen without the controller $\mathcal{C}_1$ being informed: $E_1 = \{\langle A_1, B_0, D_0, c, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_1, D_0, \epsilon, b^*, \epsilon, \epsilon\rangle, \langle A_1, B_2, D_0, \epsilon\ , b^*a, \epsilon, \epsilon\rangle, \langle A_1, B_3, D_0, \epsilon, b^*(a+\epsilon), d, \epsilon\rangle, \langle A_1, B_3, D_1, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle, \langle A_1, B_3, D_0, \epsilon, b^*(a+\epsilon), \epsilon, d\rangle\}$. However, after the second transition $\langle B_0, Q_{1,2}?c, B_1\rangle$, the controller $\mathcal{C}_2$ has an accurate state estimate: $E_2 = \{\langle A_1, B_1, D_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle\}$. We skip a few steps and consider the state estimates before the sixth transition $\langle D_1, Q_{3,1}!d, D_0\rangle$: E_1 is still the same, because the subsystem $\mathcal{T}_1$ did not perform any action, $E_3 = \{\langle A_1, B_3, D_1, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle\}$, and we do not give E_2, because $\mathcal{T}_2$ is no longer involved. When $\mathcal{T}_3$ sends message d to $\mathcal{T}_1$, it tags it with E_3. Thus, $\mathcal{C}_1$ knows, after receiving this message, that there is a message a in the queue $Q_{2,1}$. It thus disables the action $A_2 \xrightarrow{Q_{1,2}!d} A_0$, as long as this message a is not read (action $A_2 \xrightarrow{Q_{2,1}?a} A_2$), to prevent the system from reaching the forbidden states. Note that if we consider the sequence of actions of Figure 3.10 without the sending and the reception of the message a, then when $\mathcal{T}_1$ reaches the location A_2 by executing the action $Q_{3,1}?d$, its controller $\mathcal{C}_1$ enables the actions $Q_{1,2}!d$, because it knows that no message a is in $Q_{2,1}$.

3.3.4.2 Effective Algorithm for the Distributed Problem

The algorithms described in the previous sections require the implementation of (co-) reachability operators. Those operators cannot be implemented exactly because of undecidability reasons. Abstract interpretation-based techniques [CC77] allows us to implement, in a finite number of steps, an *overapproximation* of the (co-)reachability operators, and thus of the set $I(Bad)$, and of the state estimates E_i.

Computation of (Co-)Reachability Sets by Abstract Interpretation For a given set of global states $X' \subseteq X$ and a given set of transitions $\Delta' \subseteq \Delta$, the reachability (resp. co-reachability) set from X' can be characterized by the least fixpoint $\mathsf{Reach}^{\mathcal{T}}_{\Delta'}(X') = \mu Y.F_{\Delta'}(Y)$ with $F_{\Delta'}(Y) = X' \cup \mathrm{Post}^{\mathcal{T}}_{\Delta'}(Y)$ (resp. $\mathrm{Coreach}^{\mathcal{T}}_{\Delta'}(X') = \mu Y.F_{\Delta'}(Y)$ with $F_{\Delta'}(Y) = X' \cup \mathrm{Pre}^{\mathcal{T}}_{\Delta'}(Y)$). Abstract interpretation provides a theoretical framework to compute efficient over-approximations of such fixpoints. The concrete domain i.e., the sets

of states 2^X, is substituted by a simpler abstract domain Λ, linked by a *Galois connection* $2^X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Lambda$ [CC77], where α (resp. γ) is the abstraction (resp. concretization) function. The fixpoint equation is transposed into the abstract domain. So, the equation to solve has the form: $\lambda = F^\sharp_{\Delta'}(\lambda)$, with $\lambda \in \Lambda$ and $F^\sharp_{\Delta'} \sqsupseteq \alpha \circ F_{\Delta'} \circ \gamma$ where $\sqsupseteq$ is the comparison operator in the abstract lattice. In that setting, a standard way to ensure that this fixpoint computation converges after a finite number of steps to some overapproximation λ_∞, is to use a *widening operator* ∇. The concretization $c_\infty = \gamma(\lambda_\infty)$ is an overapproximation of the least fixpoint of the function $F_{\Delta'}$.

Choice of the Abstract Domain In abstract interpretation-based techniques, the quality of the approximation we obtain depends on the choice of the abstract domain Λ. In our case, the main issue is to abstract the content of the FIFO channels. Since the CFSM model is Turing-powerful, the language which represents all the possible contents of the FIFO channels may be recursively enumerable. As discussed in [LGJJ06], a good candidate to abstract the contents of the queues is to use the class of regular languages, which can be represented by finite automata. Let us recall the main ideas of this abstraction.

Finite Automata as an Abstract Domain We first assume that there is only one queue in the distributed system $\mathcal{T}$; we explain later how to handle a distributed system with several queues. With one queue, the concrete domain of the system $\mathcal{T}$ is defined by $X = 2^{L \times M^*}$. A set of states $Y \in 2^{L \times M^*}$ can be viewed as a map $Y : L \mapsto 2^{M^*}$ that associates a language $Y(\ell)$ with each location $\ell \in L$; $Y(\ell)$ therefore represents the possible contents of the queue in the location ℓ. In order to simplify the computation, we substitute the concrete domain $\langle L \mapsto 2^{M^*}, \subseteq \rangle$ by the abstract domain $\langle L \mapsto \mathrm{Reg}(M), \sqsubseteq \rangle$, where $\mathrm{Reg}(M)$ is the set of *regular languages* over the alphabet M and $\sqsubseteq$ denotes the natural extension of the set inclusion to maps. This substitution consists thus in abstracting, for each location, the possible contents of the queue by a regular language. Regular languages have a canonical representation given by finite automata, and each operation (union, intersection, left concatenation,...) in the abstract domain can be performed on finite automata.

Widening Operator With our abstraction, the widening operator we use to ensure the convergence of the computation, is also performed on a finite automaton, and consists in quotienting the nodes[10] of the automaton by the *k-bounded bisimulation relation* $\equiv_k$; $k \in \mathbb{N}$ is a parameter which allows us to tune the precision: increasing k improves the quality of the abstractions in general. Two nodes are equivalent w.r.t. $\equiv_k$ if they have the same outgoing path (sequence of labeled transitions) up to length k. While we merge the equivalent nodes, we keep all transitions and obtain an automaton recognizing a larger language. Note that the number of equivalent classes of the k-bounded bisimulation relation is bounded by a function of k and of the size of the alphabet of messages. Therefore the number of states of the resulting automaton is also bounded. So, if we fix k and we apply this widening operator regularly, the fixpoint computation terminates (see [LGJJ06] for more details and examples).

[10]The states of an automaton representing the queue contents are called nodes to avoid the confusion with the states of a CFSM.

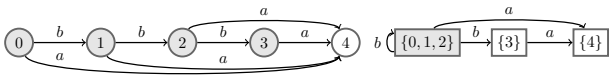


Figure 3.11: Automaton $\mathcal{A}$ and $\mathcal{A}'$ built from $\mathcal{A}$ with the 1-bounded bisimulation relation $\equiv_1$

Example 3.6 We consider the automaton $\mathcal{A}$ depicted in Figure 3.11, whose recognized language is $a + ba + bba + bbba$. We consider the 1-bounded bisimulation relation i.e., two nodes of the automaton are equivalent if they have the same outgoing transitions. So, nodes $0, 1, 2$ are equivalent, since they all have two transitions labeled by a and b. Nodes 3 and 4 are not equivalent to any other nodes since 4 has no outgoing transition whereas only a is enabled in node 3. When we quotient $\mathcal{A}$ by this equivalent relation, we obtain the automaton $\mathcal{A}'$ on the right of Figure 3.11, whose recognized language is b^*a. $\diamond$

When the system contains several queues $Q = \{Q_1, \ldots, Q_r\}$, their content can be represented by a concatenated word $w_1 \sharp \ldots \sharp w_r$ with one w_i for each queue Q_i and $\sharp$, a delimiter. With this encoding, we represent a set of queue contents by a finite automaton of a special kind, namely a QDD [BGWW97]. Since QDDs are finite automata, classical operations (union, intersection, left concatenation,...) in the abstract domain are performed as previously. We must only use a slightly different widening operator not to merge the different queue contents [LGJJ06].

Effective Algorithm The Reach and $\mathrm{Coreach}$ operators are computed using those abstract interpretation techniques: we proceed to an iterative computation in the abstract domain of regular languages and the widening operator ensures that this computation terminates after a finite number of steps [CC77]. So the Reach (resp. $\mathrm{Coreach}$) operators always give an overapproximation of the reachable (resp. co-reachable) states, whatever the distributed system is. Finally, we define the distributed controller as in section 3.3.4.1 by using the overapproximations $I'(Bad)$ and E'_i instead of $I(Bad)$ and E_i.

3.3.5 Conclusion

We propose in this section a novel framework for the control of distributed systems modeled as communicating finite state machines with reliable unbounded FIFO channels. Each local controller can only observe its subsystem but can communicate with the other controllers by piggy-backing extra information, such as state estimates, to the messages sent in the FIFO channels. Our algorithm synthesizes the local controllers that restrict the behavior of a distributed system in order to satisfy a global state avoidance property, e.g. to ensure that an error state is no longer reachable or to bound the size of the FIFO channels. We abstract the content of the FIFO channels by the same regular representation as in [LGJJ06]; this abstraction leads to a safe effective algorithm. Even if we cannot have any theoretical guarantee about the permissiveness of the control (like a non-blocking property), we remind that this permissiveness depends on the quality of the abstraction. The more precise the abstraction is, the more permissive the control is.

3.4 General Conclusion and Future Work

In this chapter, we investigated the Supervisory Control of Concurrent Discrete Event Systems. We have developed algorithms that perform the controller synthesis phase by taking advantage of the structure of the plant without expanding the system, hence avoiding the state space explosion induced by the concurrent nature of the plant. So far we have been interested in the control of systems for prefix-closed specifications modeling e.g. safety properties. It would be interesting to look for results ensuring that the controlled system is non-blocking while still avoiding the computation of the whole state space. Techniques have been developed for the state avoidance control problem [C24] and have to be extended to the language-based approach. Another point of interest would be to extend theses techniques to the hierarchical model as described in [MW06, LLW05],[J16],[C23]. Another challenge is to extend control synthesis for infinite concurrent symbolic systems (i.e. systems modeled by LTS handling numerical variables) in order to be able to consider more realistic systems.

Regarding the distributed control problem, as a further work, we intend to solve the main practical drawback of our approach: we compute and send states estimates every time a message is sent. A more evolved technique would consist in the offline computation of the set of possible estimates. Estimates would be indexed in a table, available at execution time to each local estimator. A similar on-line method would be to use the memoization technique: when a state estimate is computed for the first time, it is associated with an index that is transmitted to the subsystem which records both values. If the same estimate must be transmitted, only its index can be transmitted and the receiver can find from its table the corresponding estimate. We still have to determine what is the most efficient technique, and evaluate how it improves the current implementation. We also believe that the work of decentralized control with communication [RC11] and modular control with coordinator [KvS08] might be adapted to our framework in order to reduce the communication between controllers.

Applying classical control theory may lead to strong restrictions in controlled systems. One challenge is thus to develop techniques that would lead to the synthesis of more flexible supervisors. One direction towards this “flexible” controller synthesis scheme is to merge control and diagnosis techniques. The idea would be to control as far as possible and then to diagnose whenever a fault occurs. Another possibility would be to control the system with respect to a degraded property as soon as a diagnoser raises an alarm. An interesting application domain for these techniques would be the security of networks or web-services. Other more general and long-term research perspectives concern the active control of reconfigurable systems, in which components can dynamically join or leave the system. In this context, properties are also dynamic and depend on the system configuration. We should then design on-line synthesis techniques and automatic deployment of supervisors. We shall come back to this point in the general conclusion (Chapter 5).

Chapter 4

Formal methods for the diagnosis and the control of confidential properties

Security is one of the most important and challenging aspects in designing services deployed on large open networks like Internet or mobile phones, e-voting systems etc. For such services, naturally subject to malicious attacks, methods to certify their security are crucial. In this context there has been a lot of research to develop formal methods for the design of secure systems and a growing interest in the formal verification of security properties [Low99, BAF05, BLL$^+$05] and in their model-based testing [Sch00b, LBW05, DFG$^+$06, Le 07]. Security properties are generally divided into three categories:

- *availability*: (a user can always perform the actions that are allowed by the security policy),
- *integrity*: (something illegal cannot be performed by a user), and
- *confidentiality*: (some secret information cannot be inferred by a user).

We focus here on confidentiality and especially information flow properties. We use the notion of *opacity* defined in [BKMR08] formalizing the absence of information flow, or more precisely, the impossibility for an attacker to infer the truth of a predicate (it could be the occurrence in the system of some particular sequences of events, or the fact that the system is in some particular configurations).

Consider a predicate φ over the trajectories of a system $\mathcal{G}$ and an attacker observing only a subset of the events of $\mathcal{G}$. We assume that the attacker knows the model $\mathcal{G}$.

In this context, the attacker should not be able to infer that a trajectory of G satisfies φ. The secret φ is opaque for $\mathcal{G}$ with respect to a given partial observation if for every trajectory of $\mathcal{G}$ that satisfies φ, there exists a trajectory, observationally equivalent from the attacker's point of view, that does not satisfy φ. In such a case, the attacker can never

be sure that a trajectory of $\mathcal{G}$ satisfying φ has occurred. In the sequel, we shall consider a secret φ corresponding to a set of secret states or given by a regular language. Finally, note that the definition of opacity is general enough to define other notions of information flow like trace-based non-interference and anonymity (see [BKMR08]). Note also that *secrecy* [AČZ06] can be handled as a particular case of opacity (See Remark 4.2) and thus our framework applies to secrecy as well. While usual opacity is concerned by the current disclosure of a secret, K-opacity, introduced in [SH07], additionally models secret retrieval in the past (e.g., K execution steps before) whereas initial-opacity relates to the membership of the system's initial state within a set of secret states. The system is initial-state opaque if the observer is never sure whether the system's initial state was a secret state or not [SH13].

For these different notions of opacity, besides deciding the absence of information flow in the systems, the question that naturally arises is "*what can be done to correct the system in order to preserve its opacity*". We shall consider two different kind of techniques allowing to answer to this question:

- Supervisory control theory (SCT), which restricts the system's behavior in order to preserve the secret;
- Enforcement, which inputs observable events of the systems and outputs (possibly) modified information to observers, such that the secret is preserved.

For the latter, it is either possible to dynamically change the observability status of an event, making it invisible for the attacker [J9]; or postponing the production of the event in order to avoid the disclosure of the secret [J2] [C10], or to enforce opacity by statically inserting additional events in the output behavior of the system while preserving its observable behavior [WL12]. We do not discuss in this introduction all works related to the study of opacity of a secret for a given secret but the reader can refer to [JLF16] for a complete review of the different approaches that have been proposed in the literature in the last decade.

The contents of this chapter is as follows:

- In Section 4.1, we formally introduce the notion of opacity for a secret which can be formulated either by means of languages or states.
- Section 4.2 provides a method allowing to check the opacity of a secret for a given system. It is in particular shown that this problem is PSPACE-complete [J9].
- Section 4.3 is devoted to the diagnosis of information flow [C18]. We first characterize the set of observations allowing an attacker to infer the secret information. Further, based on the diagnosis of discrete event systems, we provide necessary and sufficient conditions under which the detection and prediction of secret information flow can be ensured, and it is possible to construct a monitor allowing an administrator to detect it.
- In Section 4.4, we provide different techniques allowing to enforce such opacity on transition systems by supervisory control following the scheme of Chapter 3 [J12],[C20].

- Finally, in Section 4.5, we follow a different approach. We introduce the notion of dynamic partial observability where the set of events the user can observe changes over time. We show how to check that a system is opaque w.r.t. a dynamic observer and also address the corresponding synthesis problem: given a system $\mathcal{G}$ and secret set of states S, compute the set of dynamic observers under which S is opaque. It turns out that this problem can be reduced to a two-players safety game and that the set of such observers can be finitely represented and can be computed in EXPTIME [J9][C17].

4.1 Confidential information

There exists an information-flow whenever an attacker, denoted $\mathcal{A}$ is able to deduce confidential information on the execution of a system $\mathcal{G}$ from the observation of a subset of events $\Sigma_a \subseteq \Sigma$. Given a run of $\mathcal{G}$ corresponding to the execution of the sequence of event s, the observation of the attacker $\mathcal{A}$ is given by the static natural projection $P_{\Sigma_a}(s)$ following the architecture of Figure 4.1

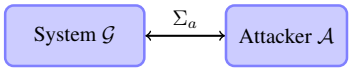


Figure 4.1: View of $\mathcal{G}$ from $\mathcal{A}$

In the sequel we let $\mathcal{G} = (Q, q_0, \Sigma, \longrightarrow)$ be a non-deterministic finite automaton over Σ. The language $\mathcal{L}_\varphi \subseteq \Sigma^*$ represents a confidential information on the execution of $\mathcal{G}$, i.e. if the current trace of a sequence is $s \in \mathcal{L}(\mathcal{G})$, the user should not be able to deduce, from the knowledge of $P_{\Sigma_a}(s)$ and the structure of $\mathcal{G}$, that $s \in \mathcal{L}_\varphi$. As stressed earlier, the attacker is armed with full information on the structure of $\mathcal{G}$ (he can perform computations using $\mathcal{G}$ like subset constructions) but has only partial observability at runtime upon its behaviors, namely the observed traces are in Σ_a^*.

Next we introduce the notion of opacity first defined in [Maz04, BKMR08]. Intuitively, a secret L_φ is *opaque* with respect to a pair $(\mathcal{G}, \Sigma_a)$ if the attacker $\mathcal{A}$ can never be sure that the current trace of the executed sequence in $\mathcal{G}$ is in $\mathcal{L}_\varphi$.

Definition 4.1 (Trace Based Opacity) *Let $\mathcal{L}_\varphi \subseteq \Sigma^*$. The secret $\mathcal{L}_\varphi$ is* opaque *with respect to $(\mathcal{L}(\mathcal{G}), \Sigma_o)$*[1] *if for all $\mu \in Traces_{\Sigma_a}(\mathcal{G})$, $[\![\mu]\!]_{\Sigma_a} \not\subseteq \mathcal{L}_\varphi$.* ■

Example 4.1 *Let $\mathcal{G}$ be the automaton depicted in Figure 4.2 with $\Sigma = \{h, p, a, b\}$, $\Sigma_a = \{a, b\}$. The secret under consideration is the occurrence of the event "h", and this can defined by $\mathcal{L}_\varphi = \Sigma^*.h.\Sigma^*$. This should not be revealed to the users of the system, knowing that "h" is not observable. $\mathcal{L}_\varphi$ is not opaque w.r.t. $(\mathcal{G}, \Sigma_a)$ as by observing "b", the sole corresponding sequence in $[\![b]\!]_{\Sigma_a}$ is $h.b$ and thus it is in $\mathcal{L}_\varphi$. Note that if the attacker observes only "a", then it cannot tell whether the current sequence of actions of the system belongs to $\mathcal{L}_\varphi$ as $[\![a]\!]_{\Sigma_a} = \{p.a, h.a\}$ and thus is not included in $\mathcal{L}_\varphi$.* □

[1] also denoted $(\mathcal{G}, \Sigma_o)$.

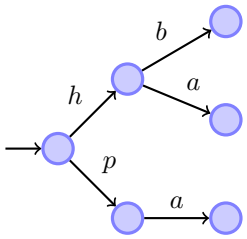


Figure 4.2: Trace based opacity illustration

Remark 4.1 *Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ in Σ^*, a secret L_φ such that $L_\varphi \subseteq \Sigma^*$ and an interface Σ_a. Assume that L_φ is opaque w.r.t. $(\mathcal{L}_1, \Sigma_a)$ and $(\mathcal{L}_2, \Sigma_a)$, then L_φ is opaque w.r.t. $\mathcal{L}_1 \cup \mathcal{L}_2$, but not necessarily w.r.t. $\mathcal{L}_1 \cap \mathcal{L}_2$.*

Given three languages $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$, $\mathcal{G}_1$, acting upon Σ such that $\mathcal{L}_1 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_3$, L_φ may be opaque w.r.t. $(\mathcal{L}_2, \Sigma_a)$ but not opaque w.r.t. $(\mathcal{L}_1, \Sigma_a)$ or $(\mathcal{L}_3, \Sigma_a)$.

In view of correcting the system whenever it is not opaque, it is interesting to characterize the sequences of $\mathcal{L}(\mathcal{G})$ that do not disclose the secret $\mathcal{L}_\varphi$.

Proposition 4.1 ([BBB$^+$07]) *Given a system $\mathcal{G}$ and a set of traces $\mathcal{L}_\varphi$, there always exists a supremal prefix-closed sublanguage $\mathcal{L}'$ of $\mathcal{L}(\mathcal{G})$ such that $\mathcal{L}_\varphi$ is opaque w.r.t. $\mathcal{L}'$ and Σ_a, namely the language $\mathcal{L}' = \mathcal{L}(\mathcal{G}) \setminus ((\mathcal{L}(\mathcal{G}) \setminus P_{\Sigma_a}^{-1} \circ P_{\Sigma_a}(\mathcal{L}(\mathcal{G}) \setminus \mathcal{L}_\varphi)).\Sigma^*)$.* ■

Intuitively, the language $P_{\Sigma_a}^{-1} \circ P_{\Sigma_a}(\mathcal{L}(\mathcal{G}) \setminus \mathcal{L}_\varphi)$ is the set of "safe" sequences that do not reveal $\mathcal{L}_\varphi$, whereas any sequence in $\mathcal{L}(\mathcal{G}) \setminus P_{\Sigma_a}^{-1} \circ P_{\Sigma_a}(\mathcal{L}(\mathcal{G}) \setminus \mathcal{L}_\varphi)$ reveals $\mathcal{L}_\varphi$ (these sequences are extended with Σ^* as, once $_\varphi$ has been revealed, this holds forever). We shall denote by $SupOp(\mathcal{L}(\mathcal{G}), \mathcal{L}_\varphi, \Sigma_0)$ the function that computes the language $\mathcal{L}'$ previously defined.
An alternative definition of opacity is when the secret is a set of states of the system.

Definition 4.2 (State Based Opacity) *Let $F \subseteq Q$. The secret F is* opaque *with respect to $(\mathcal{G}, \Sigma_o)$ if for all $\mu \in Traces_{\Sigma_a}(\mathcal{G})$, $\mathrm{Post}(\{q_0\}, [\![\mu]\!]_{\Sigma_a}) \not\subseteq F$.* ■

Remark 4.2 *We can extend the definition of opacity to a (finite) family of sets $\mathcal{F} = \{F_1, F_2, \cdots, F_k\}$: the secret $\mathcal{F}$ is* opaque *with respect to $(\mathcal{G}, \Sigma_o)$ if for all $F \in \mathcal{F}$, for all $\mu \in Traces_{\Sigma_a}(\mathcal{G})$, $\mathrm{Post}(\{q_0\}, [\![\mu]\!]_{\Sigma_a}) \not\subseteq F$. This can be used to express other kinds of confidentiality properties. For example, [AČZ06] introduced the notion of* secrecy *of a set of states F. Intuitively, F is not secret w.r.t. $\mathcal{G}$ and Σ_a whenever after an observation μ, the attacker either knows that the system is in F or knows that it is not in F. Secrecy can thus be handled considering the opacity w.r.t. a family $\{F, Q \setminus F\}$.*

In the sequel we consider only one set of states F and, when necessary, we point out what has to be done for solving the problems with a family of sets.

Example 4.2 *Consider the automaton $\mathcal{G}$ of Figure 4.3, with $\Sigma_a = \Sigma = \{a, b\}$. The secret is given by the states represented by red squares i.e., $F = \{q_2, q_5\}$. The secret F is*

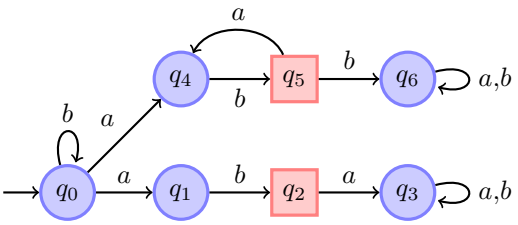


Figure 4.3: State based opacity illustration

certainly not state-based opaque with respect to $(\mathcal{G}, \Sigma)$*, as by observing a trace of* $b^*.a.b$*, an attacker* $\mathcal{A}$ *knows that the system is in a secret state. Notice that he does not know whether it is* q_2 *or* q_5 *but still he knows that the state of the system is in* F*.* □

From Trace-Based to State-Based Opacity. Let $\mathcal{L}_\varphi$ be given by a finite and complete automaton $\mathcal{A}_\varphi$ with accepting states Q_φ and initial state q_0^φ. Define $\mathcal{G} \times A_\varphi$ with accepting states $F_\varphi = Q \times Q_\varphi$.

Proposition 4.2 *If* $\mathcal{A}_\varphi$ *is deterministic then* $\mathcal{L}_\varphi$ *is trace-based opaque w.r.t.* $(\mathcal{G}, \Sigma_a)$ *iff* F_φ *is State-Based opaque w.r.t.* $(\mathcal{G} \times \mathcal{A}_\varphi, \Sigma_a)$.

Hence in this case, Trace-Based Opacity can be reduced in polynomial time to State-Based Opacity. If $\mathcal{A}_\varphi$ is non-deterministic, prior to the previous construction we need to determinize $\mathcal{A}_\varphi$ and the product has size exponential in $|\mathcal{A}_\varphi|$.

4.2 Checking State Based Opacity

In this section, without lost of generality, we shall consider the verification of the state-based opacity. Let $\mathcal{G} = (Q, q_0, \Sigma, \longrightarrow)$ be a non-deterministic LTS with F the set of secret states. Checking the opacity of F w.r.t. $(\mathcal{G}, \Sigma_a)$ is based on the determinization via subset construction adapted to our definition of opacity: $Det_{\Sigma_a}(\mathcal{G})$ denotes the deterministic LTS which is computed from $\mathcal{G}$. Now, based on this operation, to check whether F is opaque w.r.t. $(\mathcal{G}, \Sigma_a)$ we can proceed as follows:

1. determinize $\mathcal{G}$ using the construction of Definition (1.5) and denote $Det_{\Sigma_a}(\mathcal{G})$ the obtained LTS with states in 2^Q. Note that the set of accepting states is $F_o = 2^F$;
2. check whether F_o is reachable in $Det_{\Sigma_a}(\mathcal{G})$
   - if yes, then F is not opaque w.r.t. $(\mathcal{G}, \Sigma_a)$,
   - otherwise, F is opaque w.r.t. $(\mathcal{G}, \Sigma_a)$.

Indeed, if there exists $\mu \in \Sigma_a^*$ such that $\mu \in \mathcal{L}_{F_o}(Det_{\Sigma_a}(\mathcal{G}))$, then according to Def. (1.5), it entails that $\mathrm{Post}(\{q_o\}, [\![\mu]\!]_{\Sigma_a}) \subseteq F$, meaning that F is not opaque w.r.t. $(\mathcal{G}, \Sigma_a)$. Thus, according to this construction, the set of observed traces for which the attacker knows that the current execution discloses the secret is given by $\mathcal{L}_{F_o}(Det_{\Sigma_o}(\mathcal{G}))$ where $F_o = 2^F$.

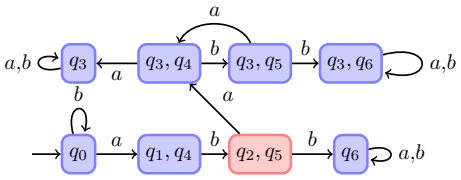


Figure 4.4: $Det_{\Sigma_a}(\mathcal{G})$

Example 4.3 *Back to the LTS $\mathcal{G}$ of Fig. 4.2, $Det_{\Sigma_a}(\mathcal{G})$ is given in Fig. 4.4. It is easy to check that $F_o = \{(q_2, q_5)\}$ is reachable from q_o, thus F is not opaque w.r.t. $(\mathcal{G}, \Sigma_a)$.*

Remark 4.3 *To check opacity for a family $\{F_1, F_2, \cdots, F_k\}$, we define $\mathcal{F}$ to be the set $2^{F_1} \cup 2^{F_2} \cup \cdots \cup 2^{F_k}$ (as pointed out before, this enables us to handle secrecy).*

The previous construction shows that State Based Opacity on non-deterministic LTS can be checked in exponential time. Actually, checking state based opacity for (non-deterministic) LTS is PSPACE-complete. Given an LTS $\mathcal{G}$ over Σ and F the set of accepting states, the (language) universality problem is to decide whether $\mathcal{G}$ accepts all possible sequences, namely if $\mathcal{L}_F(\mathcal{G}) = \Sigma^*$. If not, then $\mathcal{G}$ is not universal. From [MS72], checking language universality for a non-deterministic LTS with accepting states is PSPACE-complete. It might be shown that the problems of the problem of checking opacity of a secret and the (language) universality problem the are equivalent [J9]. We thus obtain the following result:

Theorem 4.1 *Checking opacity of F w.r.t. $(\mathcal{G}, \Sigma_a)$ is PSPACE-complete for non-deterministic LTS.*

Whenever $\mathcal{G}$ is not opaque, it is interesting to be able to detect an information flow from an attacker $\mathcal{A}$. This is the aim of the next section.

4.3 Diagnosis of information flows

We here consider three components: a system $\mathcal{G}$, an attacker $\mathcal{A}$ and a monitor $\mathcal{M}$ (modelling for example the administrator of the system or an intrusion detection system) (C.f. Figure 4.5). We assume that the system $\mathcal{G}$ is modeled by a finite transition system. Users interact with $\mathcal{G}$ through an interface P_{Σ_o}, corresponding to the inputs/outputs of the system.

For this system, one can define some confidentiality policies. Following the approach of [C22] for the diagnosis and [AČZ06, BKMR08], a secret is modeled by a property φ given as a regular language over the alphabet Σ of the system $\mathcal{G}$. The secret is preserved as far as the attacker cannot surely infer that the property φ is satisfied by the current execution of the system based on the observations performed through the interface P_{Σ_o}. *A contrario*, the monitor $\mathcal{M}$ tries to analyze the information flow between the system $\mathcal{G}$ and the attacker $\mathcal{A}$ in order to raise an alarm whenever the secret has been revealed.
$\mathcal{M}$ can also try to predict the information flow. To do so, we assume that $\mathcal{M}$ knows the power of the attacker (i.e. he knows the model of the system $\mathcal{G}$ and the interface P_o of the attacker). He observes the system through the interface P_{Σ_m} (we do not assume any link

Figure 4.5: Architecture

between the two interfaces). Further, based on the set of observations allowing the attacker to infer the secret information, we provide necessary and sufficient conditions under which detection and prediction of secret information flow can be ensured, and construct a monitor $\mathcal{M}$ allowing an administrator to detect the attacks. This supervision is performed on-line, the monitor raising an alarm whenever an information flow occurs.

4.3.1 Inference of Opacity by the attacker $\mathcal{A}$

In this section, we consider a user $\mathcal{A}$ interacting with a system modeled by an LTS $\mathcal{G} = (Q, q_o, \Sigma, \rightarrow)$ through an interface modeled by the projection P_{Σ_a} and we assume that the secret is given by a complete and deterministc LTS $\mathcal{A}_\varphi$ with accepting states Q_φ and initial state q_o^φ. We say that a trajectory $s \in \mathcal{L}(\mathcal{G})$ is recognized by $\mathcal{A}_\varphi$, noted $s \models \varphi$ whenever $s \in \mathcal{L}_{Q_\varphi}(\mathcal{A}_\varphi)$.

Next, following the results presented in Section 2.2, we can build a function $\mathcal{O}_\mathcal{A}^\varphi$ which gives access, for each observation $\mu \in \mathit{Traces}_{\Sigma_a}(\mathcal{G})$ to what an attacker $\mathcal{A}$ can infer on s and φ. Formally, if s is the current execution of the system and $\mu = P_{\Sigma_a}(s)$ is the corresponding observation, the verdicts we are interested in are defined by the following function:

$$\mathcal{O}_\mathcal{A}^\varphi : \Sigma_\mathcal{A}^* \rightarrow V = \{Yes, Inev, Inev\_Yes, Never, No, ?\}$$

where the semantic of the verdicts is as follows:

1) $\mathcal{O}_\mathcal{A}^\varphi(\mu) = Yes$ if $\mathcal{A}$ knows that for the current execution s (s.t. $P_\mathcal{A}(s) = \mu$), $s \models \varphi$;

2) $\mathcal{O}_\mathcal{A}(\mu) = Inev$ if $\mathcal{A}$ knows that $s \not\models \varphi$ but also that φ will be inevitably satisfied by all the possible extensions of s;

3) $\mathcal{O}_\mathcal{A}^\varphi(\mu) = Inev\_\varphi$ if $\mathcal{A}$ knows that $s \models \varphi$ or that φ will inevitably be satisfied in the future but cannot distinguish between the two cases so far

4) $\mathcal{O}_\mathcal{A}^\varphi(\mu) = No$ if $\mathcal{A}$ knows that $s \not\models \varphi$, but φ is neither unavoidable nor impossible;

5) $\mathcal{O}_\mathcal{A}^\varphi(\mu) = \ ?$ in all the other cases, meaning that $\mathcal{A}$ cannot infer any useful information with regards to s and φ after the observation $\mu = P_\mathcal{A}(s)$.

The explanation of how such a function can be derived from $\mathcal{G}$, $\mathcal{A}$ and Σ_a as given in Section 2.2.3.

Example 4.4 *Consider the system $\mathcal{G}$ described in the left-hand side of Fig. 4.6. The alphabet of $\mathcal{G}$ is $\Sigma = \{a, b, c, X, Y, Z, h, \tau, \delta\}$. We assume here that the secret property is given by the occurrence of h (LTS $\mathcal{A}_\varphi$ in the right-hand side of Fig. 4.6); in this example, the attacker tries to infer the occurrence of the event h in the system.*
The interface of the attacker is reduced to $\Sigma_\mathcal{A} = \{a, b, c, \delta\}$. The observer $\mathcal{O}_\mathcal{A}^\varphi$ that the attacker $\mathcal{A}$ can build is given by the LTS depicted in the lower part of Fig. 4.6.

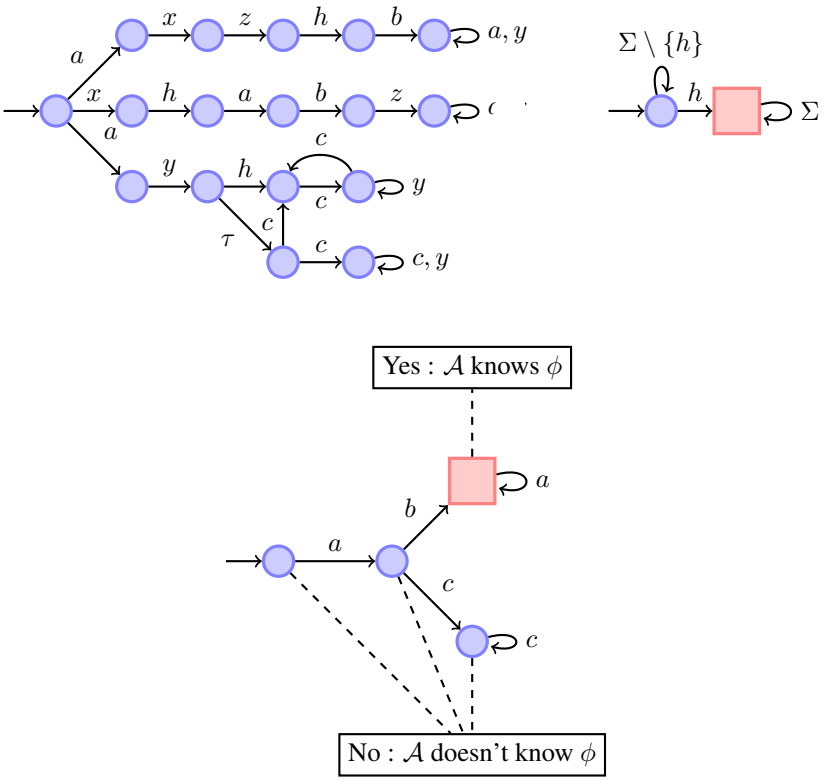


Figure 4.6: $\mathcal{G}$, $\mathcal{A}_\varphi$ and the corresponding $\mathcal{O}_{\mathcal{A}}^{\varphi}$

4.3.2 Monitoring Opacity

Given a secret φ, based on the techniques described in the preceding sections, it is possible to check whether φ is opaque w.r.t. $\mathcal{G}$ and the interface P_{Σ_a}. When φ is not opaque, it can be important for an administrator to supervise the system on-line by means of a monitor $\mathcal{M}$ and to raise an alarm as soon as an information flow occurs. For this, we assume that $\mathcal{M}$ knows the model of the system $\mathcal{G}$ and observes it through the interface P_{Σ_m}. Moreover, $\mathcal{M}$ knows the ability of the attacker $\mathcal{A}$, meaning that the monitor knows that $\mathcal{A}$ observes the system via the interface P_{Σ_a} and that he can construct an observation function $\mathcal{O}_{\mathcal{A}}^{\varphi}$. We do not assume any relation between P_{Σ_a} and P_{Σ_m}. Thus, $\mathcal{M}$ has to infer the attacker's knowledge based on the observation of $\mathit{Traces}_{\Sigma_m}(\mathcal{G}) \subseteq \Sigma_m^*$. If φ is not opaque w.r.t. the system $\mathcal{G}$ and the interface P_{Σ_a}, an administrator can build an observation function to diagnose the fact that the secret has been revealed. One can also be more accurate and try to predict the fact that the secret will be inevitably known by the attacker strictly before the information flow.

Note that it is not necessary to diagnose the fact that the system performed a sequence satisfying the secret if this sequence does not correspond to a non-opaque execution (this sequence does not reveal anything to the attacker); only the executions that lead to an information flow have to be taken into account. Indeed, the secret φ is revealed to the attacker by an execution $s \in \mathcal{L}(\mathcal{G})$ if and only if $P_{\Sigma_a}(s) \in \mathcal{L}_{F_0}(Det_{\Sigma_a}(\mathcal{G} \times A_\varphi))$.

In other words, we are interested in diagnosing the property: "The secret φ has been revealed to the attacker", which corresponds to the extention-closed language that can be recognized by an LTS Ω, equipped with a set of final states F_Ω such that:

$$\mathcal{L}_{F_\Omega}(\Omega) = P_{\Sigma_a}^{-1}(\mathcal{L}_{F_0}(Det_{\Sigma_a}(\mathcal{G} \times A_\varphi))) \cdot \Sigma^* \tag{4.1}$$

Example 4.5 *To illustrate the computation of (4.1), let us come back to Example 4.4. It is given by Fig. 4.7.*

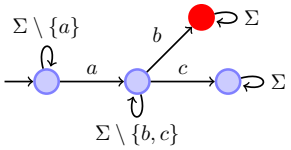


Figure 4.7: The LTS Ω computed from $Det_{\Sigma_a}(\mathcal{G} \times A_\varphi)$

Supervision of Information Flow Given a system $\mathcal{G}$, an attacker $\mathcal{A}$ observing $\mathcal{G}$ via the interface P_{Σ_a} and a secret φ (that we assume to be non-opaque), we describe now a method allowing an administrator $\mathcal{M}$ observing $\mathcal{G}$ via the interface P_{Σ_m} to know whether there is an information flow or not. We assume that the monitor in charge of the supervision has a full knowledge of $\mathcal{G}$ and knows the observation mask P_{Σ_a}. As mentioned in the introduction of this section, $\mathcal{M}$ does not directly observe φ. Only the trajectories causing an information flow have to be supervised. We consider then the stable property Ω corresponding to the trajectories of $\mathcal{G}$ inducing an information flow from $\mathcal{G}$ to $\mathcal{A}$ (see Eq. (4.1)). In order to

construct the observer $\mathcal{O}^{\Omega}_{\mathcal{M}}$ in charge of the supervision of Ω (i.e. corresponding to the information leak of φ), we first build $\mathcal{G}_{\Omega} = \mathcal{G} \times \Omega$ and the sets $F_{\mathcal{G}_{\Omega}}$, $I_{\mathcal{G}_{\Omega}}$, $N_{\mathcal{G}_{\Omega}}$ (as described in Step 2., Section 2.2.3).

Now, based on the techniques of the section 2.2.3, one can compute the LTS $Det_{\Sigma_a}(\mathcal{G}_{\Omega})$ over Σ_{Σ_m} from which we can derive an observer $\mathcal{O}^{\Omega}_{\mathcal{M}}$ with the following verdicts: for $\mu \in \mathit{Traces}_{\Sigma_m}(\mathcal{G})$,

- $\mathcal{O}^{\Omega}_{\mathcal{M}}(\mu) =$ "Yes": $\mathcal{M}$ infers that Ω is satisfied and thus can deduce that $\mathcal{A}$ knows φ;
- $\mathcal{O}^{\Omega}_{\mathcal{M}}(\mu) =$ "No": $\mathcal{M}$ knows that $\mathcal{A}$ does not know φ but might know it in the future;
- $\mathcal{O}^{\Omega}_{\mathcal{M}}(\mu) =$ "$Inev$": $\mathcal{M}$ knows that $\mathcal{A}$ will inevitably know φ but does not know it yet;
- $\mathcal{O}^{\Omega}_{\mathcal{M}}(\mu) =$ "$Inev\_Yes$": $\mathcal{M}$ knows that $\mathcal{A}$ already knows or will know φ;
- $\mathcal{O}^{\Omega}_{\mathcal{M}}(\mu) =$ "?" means that $\mathcal{M}$ cannot deduce anything about the knowledge of $\mathcal{A}$.

Unfortunately, the case $\mathcal{O}^{\Omega}_{\mathcal{M}}(\mu) =$ "?" does not imply that the attacker $\mathcal{A}$ does not know φ. As $\mathcal{M}$ and $\mathcal{A}$ observe the system via different interfaces, it might be the case that $\mathcal{A}$ already knows φ and that $\mathcal{M}$ will never infer this information. This corresponds to the non-diagnosability of Ω [C22]. This can occur when there exist two arbitrarily long trajectories s and s' corresponding to the same observation μ such that $s \in \mathcal{L}_{F_{\Omega}}(\Omega)$ (thus a non-opaque trajectory of φ) and $s' \notin \mathcal{L}_{F_{\Omega}}(\Omega)$. In the next section, we will give necessary and sufficient conditions under which this case does not occur.

4.3.3 Necessary and sufficient conditions for detection/prediction of information flow

Consider the system $\mathcal{G}$ as well as the property Ω described in the previous section.

4.3.3.1 Diagnosability

Intuitively, $\mathcal{G}$ is Ω-diagnosable ([SSL$^+$96],[C22]) if there exists $n \in \mathbb{N}$ such that for any trajectory s of $\mathcal{G}$ such that $s \models \Omega$, Ω becomes non-opaque after waiting for at most n observations. In the case of monitoring opacity, this means that when the monitor is observing a trace in $\mathcal{L}_{F_{\Omega}}(\Omega)$, a "Yes" answer should be produced by the observer after finitely many observed events. Hence, if there exists $s \in \mathcal{L}(\mathcal{G})$ triggered by the system such that s is non-opaque for $\mathcal{A}$, then $\mathcal{M}$ will surely know it at most n observed events after the observation of $P_{\Sigma_m}(s)$.

4.3.3.2 Predictability

If the system is Ω-diagnosable, then it might be interesting to refine the verdict by predicting the satisfaction of the property strictly before its actual occurrence (see Section 2.2). Roughly speaking, Ω is predictable if it is always possible to detect the future satisfaction of Ω, strictly before this happens, only based on the observations. This property means that for any trajectory s that satisfies Ω, there exists a strict prefix t that does not satisfy Ω, such

that any trajectory u compatible with observation $P_{\Sigma_m}(t)$ will inevitably be extended into a trajectory $u.v$ satisfying Ω. In our setting, this means that $\mathcal{M}$ can always predict that $\mathcal{A}$ will know φ and then the system operator can be warned in time to halt the system or can take counter-measures in order to avoid the secret to be revealed.

Example 4.6 *To illustrate this section, we still consider the system $\mathcal{G}$ and the secret φ defined in Example 4.4. Assume that the interface of the monitor $\mathcal{M}$ is $\Sigma_m = \{x, y\}$,*

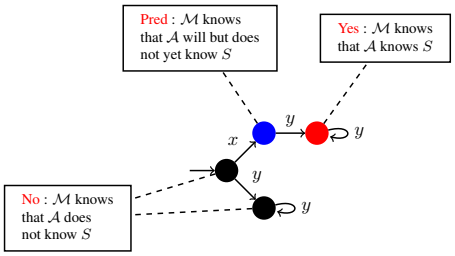


Figure 4.8: Observation function $\mathcal{O}^{\Omega}_{\mathcal{M}}$

then the system is Ω-predictable. Indeed, after the observation of x, $\mathcal{M}$ knows that all the possible extensions will satisfy Ω and thus that the secret will be revealed.

4.3.4 Conclusion

So far, in Section 4.2 and Section 4.3, we have shown how to model-check the opacity of a system and to diagnose/predict that a user considered as an attacker might have inferred information regarding the secret φ of the system $\mathcal{G}$. In the next sections, we will show how to "correct" the system in such a way that this leakage of information does not happen anymore.

4.4 Ensuring opacity by control

Our aim in this section is not to model-check (Section 4.2) or to diagnose information flow (Section 4.3), but to enforce such properties on transition systems by supervisory control following the scheme of Chapter 3. In these works, one considers a finite and deterministic labeled transition system $\mathcal{G}$ over an alphabet Σ, a regular set $S \subseteq \Sigma^*$ (the secret) and a subset $\Sigma_a \subseteq \Sigma$ (the alphabet of the adversary), and one searches for a finite state supervisor $\mathcal{C}$ enforcing the opacity of S w.r.t. the natural projection from Σ^* to Σ_a^* (by the opacity of $\mathcal{L}_\varphi$ we mean the opacity of the set of runs with traces in $\mathcal{L}_\varphi$). In this setting, $\mathcal{G}$ represents a system running in the scope of an inquisitive adversary. The adversary observes all actions in Σ_a and tries to infer from these partial observations the knowledge that the trace of the run of $\mathcal{G}$ is in the secret set $\mathcal{L}_\varphi$.

Figure 4.9: Control Architecture

4.4.1 The opacity problem and preliminary results

One subtlety lays in the additional assumption that the adversary knows exactly the system $\mathcal{G}$ and the supervisor $\mathcal{C}$. This means that new confidential information may be inferred by the adversary from the knowledge of $\mathcal{C}$ and the partial observation of the run of the controlled system. To solve the problem, one might think of iterating the construction, thus producing a decreasing chain of regular languages $\mathcal{L}(\mathcal{G}) = \mathcal{K}_0 \supseteq \mathcal{K}_1 \supseteq \mathcal{K}_1 \supseteq \dots$. Unfortunately, the iteration may be infinite, hence it may not yield an effective construction and it does not show either that this limit is regular. The exact problem is to find a supervisor $\mathcal{C}$ that enforces the opacity of $\mathcal{L}_\varphi$ w.r.t. $\mathcal{L}(\mathcal{G}/\mathcal{C})$ and the projection from Σ^* to Σ_a^*, and this is an intrinsically circular problem because the control objective cannot be expressed without an explicit reference to the controller (this is not true when the control objective is a safety property). The control problem can be stated as follows:

Problem 4.1 *Given a system $\mathcal{G}$ and a set of traces $\mathcal{L}_\varphi$. an Attacker $\mathcal{A}$ observing the system through $\Sigma_a \subseteq \Sigma$. Check the existence of a maximally permissive controller $\mathcal{C}$ that observes $\mathcal{G}$ trough the interface Σ_m and controlling $\Sigma_c \subseteq \Sigma_m$ such that $\mathcal{L}_\varphi$ is opaque w.r.t. $(\mathcal{L}(\mathcal{G}/\mathcal{C}), \Sigma_a)$.*

In order to solve this problem, we first adapt a language-based approach and introduce the following set of languages:

$$\mathcal{K}_{\mathcal{L}_\varphi} = \{\mathcal{K} \subseteq \mathcal{L}(\mathcal{G}) \mid \begin{array}{l} \mathcal{K} \text{ is controllablle w.r.t. } \mathcal{L}(\mathcal{G}) \text{ and } \Sigma_c, \\ \mathcal{K} \text{ is observable w.r.t. } \mathcal{L}(\mathcal{G}),\ \Sigma_m \text{ and } \Sigma_c,\ \} \\ \mathcal{L}_\varphi \text{ is opaque w.r.t. } (\mathcal{K}, \Sigma_a) \end{array}$$

and we denote by

$$\mathcal{K}^\uparrow = \cup_{\mathcal{K} \in \mathcal{K}_{\mathcal{L}_\varphi}} \mathcal{K}$$

its supremal element. It turns out that unlike the other controllability and observability properties (under the assumption that $\Sigma_c \subseteq \Sigma_m$), the opacity property is stable under arbitrary union. This implies the following result:

Proposition 4.3 *Problem 4.1 has a solution if $\mathcal{K}^\uparrow \neq \emptyset$, otherwise no control can enforce the opacity of $\mathcal{L}_\varphi$.*

Even though the previous proposition entails the existence of a unique maximal sublanguage of $\mathcal{L}(\mathcal{G})$, that is controllable, observable and in restriction to which $\mathcal{L}_\varphi$ is opaque, we still have to examine whether this language is regular (or at least, to exhibit sufficient conditions for regularity) and to provide an effective computation of this language. It may be remarked that restricting languages to ensure controllability and observability does not always preserve opacity and the other way round (see Example 4.7). Thus, in a first attempt

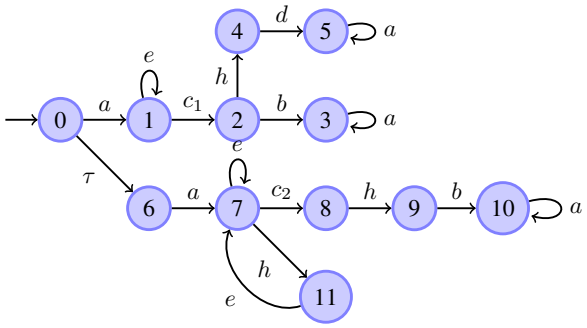


Figure 4.10: Control of non-opacity (I)

towards an effective computation of $\mathcal{K}^{\uparrow}$, following the classical methodology of Supervisory Control Theory, we establish below a fix-point characterization of this language by alternating the computation of the supremal sub-language that ensures the opacity of $\mathcal{L}_{\varphi}$ and the supremal controllable and observable sub-language. To do so, consider the following operator:

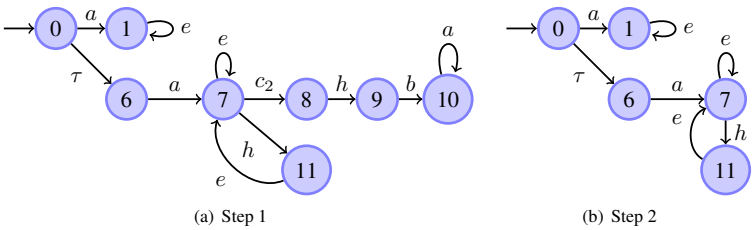
$$Sup\mathcal{K}(\bullet) = SupCo(SupOp(\bullet, \mathcal{L}_{\varphi}, \Sigma_0), \mathcal{L}(\mathcal{G}), \Sigma_c, \Sigma_m)$$

Remark that $Sup\mathcal{K}(\bullet)$ si monotonic w.r.t. set inclusion. Now, as the prefix-closed subsets of $\mathcal{L}(\mathcal{G})$ form a complete sub-lattice of $\mathcal{P}(\Sigma^*)$, it follows from Tarski's Theorem [Tar55] that $Sup\mathcal{K}(\bullet)$ has a greatest fix-point in this sub-lattice. Let $\mathcal{K}(\mathcal{L}_{\varphi}, \mathcal{L}(\mathcal{G}))$ be the greatest fix-point of the operator $Sup\mathcal{K}(\bullet)$ included in $\mathcal{L}(\mathcal{G})$. We can prove that

Proposition 4.4 $\mathcal{K}(\mathcal{L}_{\varphi}, \mathcal{L}(\mathcal{G})) = \mathcal{K}^{\uparrow}$

This fix-point computation is illustrated in Example 4.7.

Example 4.7 *The system to be controlled is given in Figure 4.10. We assume that* $\Sigma_a = \{a, b, d, e\}$, $\Sigma_m = \{a, c_1, c_2, b, d, e\}$, *and* $\Sigma_c = \{b, c_1, c_2, e\}$. *The secret is given by the language* $\mathcal{L}_{\varphi} = \Sigma^*.h.\Sigma^*$. *When observing* d, *the attacker knows that* h *occurred and the secret is revealed (in state 5). By control, action* c_1 *is disabled, thus avoiding the uncontrollable sequence* $h.d$ *to be triggered, and the LTS depicted in Figure 4.11(a) is obtained. However, doing so, the secret is now revealed to the attacker who knows the*



(a) Step 1 (b) Step 2

Figure 4.11: Control of non-opacity (II)

control law after the observation of the action b, which leads to disable the action c_2, giving the LTS of Figure 4.11(b). The secret is now opaque with respect to this LTS, which is the maximal sub-LTS of the system with this property. ◇

Note that this fix-point characterization of $\mathcal{K}^{\uparrow}$ does not ensure that this language can be always computed by a finite iteration as shown in Example 2 in [C20].

4.4.2 Effective computation of the supremal solution

We now investigate sufficient conditions, induced by relations between the alphabets Σ_c, Σ_m, and Σ_a under which $\mathcal{K}^{\uparrow}$ is regular and one can effectively compute a finite automaton generating this optimal opacity control. We first describe two cases for which the previous fix-point computation terminates and finally design a novel algorithm not derived from the classical control theory that computes $\mathcal{K}^{\uparrow}$ whenever $\Sigma_a \subseteq \Sigma_m$ but Σ_c and Σ_a are not necessarily comparable.

Assumption 1: $\Sigma_c \subseteq \Sigma_m \subseteq \Sigma_a$ **[C20].** Under this assumption, the controller observes and controls only a part of the actions of the attacker, meaning that it is less powerful than the attacker. Nevertheless, this is a sufficient condition allowing to solve the control problem.

Proposition 4.5 *Assume $\Sigma_c \subseteq \Sigma_m \subseteq \Sigma_a$, then*

$$\mathcal{K}^{\uparrow} = Sup\mathcal{K}(\mathcal{L}(\mathcal{G}))$$

Assumption 2: $\Sigma_a \subseteq \Sigma_c \subseteq \Sigma_m$ **[C20].** This assumption simply means that the controller can observe all the actions of the attacker and control them.

Proposition 4.6 *Assume $\Sigma_a \subseteq \Sigma_c \subseteq \Sigma_m$, then*

$$\mathcal{K}^{\uparrow} = SupOp(\mathcal{L}(\mathcal{G}), \mathcal{L}_{\varphi}, \Sigma_0)$$

In [C20], we provided non-trivial adaptation of Ramadge and Wonham's methods effectively computes $\mathcal{K}^{\uparrow}$, whenever $\Sigma_c \subseteq \Sigma_a \subseteq \Sigma_m$, meaning that all actions of the attacker can be observed by the controller, but only a part of them can be controlled[2]. One can think that the controller can filter out the requests sent by the attacker to the system, whereas the outputs of the system cannot be disabled by the controller. This is for example the behavior of a firewall for Internet services. We hereby focus on a less restrictive assumption.

Assumption 3: $\Sigma_c \subseteq \Sigma_m$ **and** $\Sigma_a \subseteq \Sigma_m$ **[J12].** We first show that solving the Opacity Control Problem under the assumption $\Sigma = \Sigma_a$ (full observation) induces a general solution of the Opacity Control Problem. The parameter Σ_a of the Opacity Control Problem will therefore be eliminated from the subsequent sections where the problem is afforded a solution. Indeed one can show that the Opacity Control Problem with parameters $(\Sigma,\ \mathcal{L}(\mathcal{G}),\ \mathcal{L}_{\varphi},\ \Sigma_c, \Sigma_a,\ \Sigma_m)$ is equivalent to the same problem with parameters

[2] In [TO08], the authors also provided a method whenever $\Sigma_m = \Sigma$ and $(\forall s, s' \in \mathcal{L}(\mathcal{G}))(\forall \sigma \in \Sigma_{uc} \cap \Sigma_a)$ $s \sim_{\Sigma_a} s' \wedge s\sigma \in \mathcal{L}(\mathcal{G}) \Rightarrow s'\sigma \in \mathcal{L}(\mathcal{G})$

$(\Sigma_m,\ P_{\Sigma_m}(\mathcal{L}(\mathcal{G})),\ \mathcal{L}'_\varphi,\ \Sigma_c, \Sigma_a,\ \Sigma_m)$, with $\mathcal{L}'_\varphi = P_{\Sigma_m}(\mathcal{L}'_\varphi) \setminus P_{\Sigma_m}(\mathcal{L}(\mathcal{G}) \setminus \mathcal{L}_\varphi)$ (Proposition5 in [J12]). Based on this, whenever $\Sigma_a \subseteq \Sigma_m \subseteq \Sigma$, one can reformulate the opacity control problem in terms of the abstract system induced by the observation map P_{Σ_m} and a new secret $\mathcal{L}'_\varphi$ derived from $\mathcal{L}_\varphi$, solve the problem in this abstract setting, and lift up the solution $\mathcal{K}'^{\uparrow}$ to the original setting as $\mathcal{K}^{\uparrow} = P^{-1}_{\Sigma_m}(\mathcal{K}'^{\uparrow}) \cap \mathcal{L}(\mathcal{G})$.

In the sequel, we assume without lost of generality that the transition system $\mathcal{G}$ recognizes the secret $\mathcal{L}_\varphi$, i.e. $G = (Q, \Sigma, , q_0, \longrightarrow, Q_\varphi)$ such that for any $s \in \Sigma^*$, $s \in \mathcal{L}(\mathcal{G})$ iff $\delta_\mathcal{G}(q_0, s)$ is defined and $s \in \mathcal{L}_\varphi$ iff $\delta_\mathcal{G}(q_0, s) \in Q_\varphi$. This second condition, even though it does not hold for arbitrary $\mathcal{G}$ and $\mathcal{L}_\varphi$, always holds for the parallel composition of $\mathcal{G}$ and a complete deterministic automaton recognizing $\mathcal{L}_\varphi$.

Throughout the section, $d : Q \times 2^Q \times \Sigma \to Q$ denotes a partial map such that $d(q, E, \sigma)$ is either equal to $\delta(q, \sigma)$ or undefined. Let us recall the following.

Definition 4.3 *Given $d : Q \times 2^Q \times \Sigma \to Q$, define inductively $d(q, E, \varepsilon) = q$ and $d(q, E, \sigma s) = d(d(q, E, \sigma), E, s)$ for $\sigma \in \Sigma_{ua}$ and $s \neq \varepsilon$. Let $\delta_d : (Q \times 2^Q) \times \Sigma \to (Q \times 2^Q)$ be the partial transition map on $Q \times 2^Q$ such that $\delta_d((q, E), \sigma) = (q', E')$ is given by $q' = d(q, E, \sigma)$, $E' = E$ if $\sigma \notin \Sigma_a$, and $E' = d(E, E, (\Sigma \setminus \Sigma_a)^*\sigma)$ otherwise. For $s \in \Sigma^*$, define inductively $\delta_d((q, E), \varepsilon) = (q, E)$ and $\delta_d((q, E), s\sigma) = \delta_d(\delta_d((q, E), s), \sigma)$. Let $\mathcal{A}(d)$ denote the LTS which is generated by the partial transition map δ_d from the initial state $(q_0, \{q_0\})$. Thus $\mathcal{A}(d) = (\Theta_d, \Sigma, \delta_d, (q_0, \{q_0\}))$ where Θ_d is the closure of the set $\{(q_0, \{q_0\})\}$ under δ_d. Finally let $\mathcal{L}(d) = \mathcal{L}(\mathcal{A}(d))$.* •

Note that by definition of δ_d, $\mathcal{L}(d) \subseteq \mathcal{L}(\mathcal{G})$.

Example 4.8 *Let $\mathcal{G}$ be the transition system depicted in Figure 4.12, with $Q = \{0, \ldots, 11\}$, $q_0 = 0$, $Q_S = \{1, 5, 7, 11\}$, and $\Sigma = \{u, d, c, s\}$.*

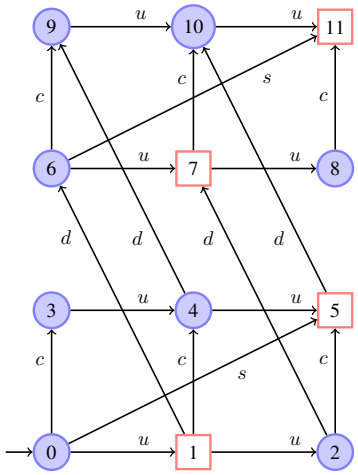


Figure 4.12: G

Let $\delta_\mathcal{G} : Q \times \Sigma \to Q$ be the partial map defined according to the arcs of the picture. $\mathcal{G}$ may

be seen as a representation of all sequences of possible moves of an agent in a three storey building with a south wing and a north wing, both equipped with lifts and both connected by a corridor at each floor. Moreover, there is a staircase that leads from the first floor in the south wing to the third floor in the north wing. The agent starts from the first floor in the south wing. He can walk up the stairs (s) or walk through the corridors (c) from south to north without any control. The lifts can be used several times one floor upward (u) and at most once one floor downwards (d) altogether. The moves of the lifts are controllable. Thus $\Sigma_c = \{u, d\}$ and $\Sigma_{uc} = \{s, c\}$. The secret is that the agent is either at the second floor in the south wing or at the third floor in the north wing (red square states in Figure 4.12). The adversary may gather the exact sub-sequence of moves in $\Sigma_a = \{s, c, u\}$ from sensors, but he cannot observe the upward moves of the lifts. Thus $\Sigma_a = \{u, c, s\}$ and $\Sigma_{uo} = \{d\}$.

Let $d_0 : Q \times 2^Q \times \Sigma \to Q$ be the map defined with $d_0(q, E, \sigma) = \delta(q, \sigma)$. The automaton $\mathcal{A}(d_0)$ defined from G is depicted in Figure 4.13. Each configuration (q, E) of the automaton is represented with q in the first line and with the list of elements of E in the second line. For instance,

$$\begin{aligned}\delta_{d_0}((1,\{1\}),c) &= (d_0(1,\{1\},c), d_0(\{1\},\{1\},d^*c))\\ &= (4,\{4,9\}).\end{aligned}$$

In fact,

- $d_0(1,\{1\},c) = \delta(1,c) = 4$,
- $d_0(1,\{1\},d) = \delta(1,d) = 6$, *and*
- $d_0(1,\{1\},dc) = d_0(d_0(1,\{1\},d),\{1\},c) = \delta(6,c) = 9$.

Similarly,

$$\begin{aligned}&\delta_{d_0}((4,\{4,9\}),u)\\ &\quad= (d_0(4,\{4,9\},u), d_0(\{4,9\},\{4,9\},d^*u)) = (5,\{5,10\})\end{aligned}$$

because

$$\begin{aligned}&d_0(4,\{4,9\},d^*u)\\ &\quad= \{d_0(4,\{4,9\},u), d_0(d_0(4,\{4,9\},d),\{4,9\},u)\}\\ &\quad= \{\delta(4,u), \delta(\delta(4,d),u)\} = \{5,10\}.\end{aligned}$$

In contrast

$$\begin{aligned}&\delta_{d_0}((4,\{4\}),d)\\ &\quad= (d_0(4,\{4\},d),\{4\}) = (\delta(4,d),\{4\}) = (9,\{4\})\end{aligned}$$

because $d \in \Sigma_{ua}$. ◇

All other examples presented in the section are continuations of Example 4.8.

Throughout the section, we let $\theta = (\widetilde{q}, \widetilde{E}) \in Q \times 2^Q$ with $\widetilde{q} \in d(\widetilde{E}, \widetilde{E}, \Sigma_{ua}^*)$.

We will now investigate which words in $\mathcal{L}(d)$ actually disclose the secret $\mathcal{L}_\varphi$ to the adversary, and how one can remedy these security failures. First, let us introduce a definition.

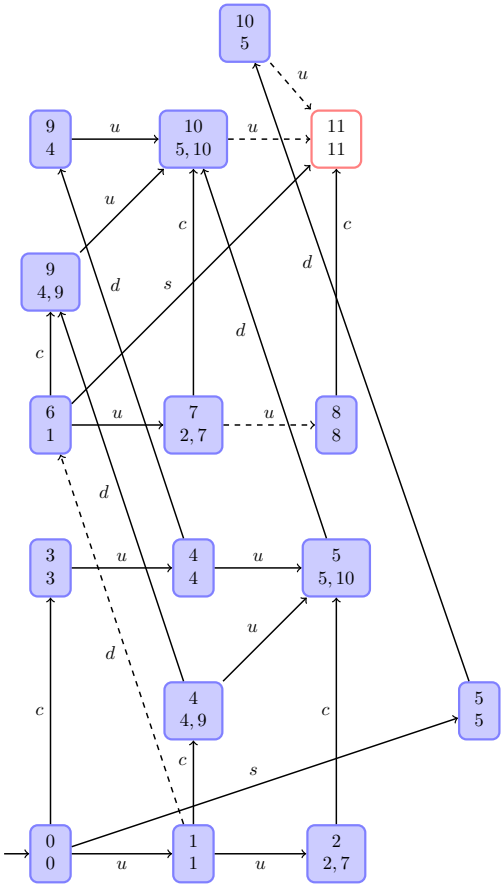


Figure 4.13: $\mathcal{A}_{d_0}$

Definition 4.4 *Given a partial map $d : Q \times 2^Q \times \Sigma \to Q$, let $\mathcal{LE}(d) = \{E \subseteq Q \mid E \neq \emptyset \wedge d(E, E, \Sigma_{uo}^*) \subseteq Q_\varphi\}$ be the associated set of* loosing estimates*, and for any $\theta = (\widetilde{q}, \widetilde{E})$ in $Q \times 2^Q$ such that $\widetilde{q} \in d(\widetilde{E}, \widetilde{E}, \Sigma_{uo}^*)$, let*

$$\mathcal{LT}(d, \theta) = \{s \in \Sigma^* \mid \delta_d(\theta, s) \in Q \times \mathcal{LE}(d)\}$$

be the set of loosing traces *w.r.t. state $\widetilde{q}$ of $\mathcal{G}$, adversary's state estimate $\widetilde{E}$ and control d.* •

Proposition 4.7 *If for every configuration $\theta = \delta_d((q_0, \{q_0\}), \widetilde{s})$ reached in $\mathcal{A}(d)$, $\mathcal{LT}(d, \theta) = \emptyset$ then $\mathcal{L}_\varphi$ is opaque w.r.t. $\mathcal{L}(d)$ and Σ_a.*

We have now in hands all elements needed to compute $d^\uparrow : Q \times 2^Q \times \Sigma \to Q$ such that $\mathcal{L}_\varphi$ is opaque w.r.t. $(\mathcal{L}(d^\uparrow), \Sigma_a)$ and $\mathcal{L}(d^\uparrow) = K^\uparrow$ is the largest controllable sublanguage of $\mathcal{L}(\mathcal{G})$ with this property. To do so, we introduce the following partial map:

Definition 4.5 *Given $d : Q \times 2^Q \times \Sigma \to Q$, let $\phi(d) \subseteq d$ be the partial map such that*

- *$\phi(d)(q, E, \sigma)$ is undefined if $\sigma \in \Sigma_c$ and*
  - *either $q \notin d(E, E, \Sigma_{ua}^*)$*
  - *or $\mathcal{LT}(d, \theta) \cap \Sigma_{uc}^* \neq \emptyset$ for $\theta = \delta_d((q, E), \sigma)$,*
- *$\phi(d)(q, E, \sigma) = d(q, E, \sigma)$ otherwise.*

Example 4.9 *Let $d_1 = \phi(d_0)$. We restrict our attention to pairs (q, E) such that $E \subseteq E'$ for some reachable configuration (q, E') of $\mathcal{A}(d_0)$, which is enough for computing $\mathcal{A}_{d_1}$. Then $d_1(q, E, u)$ is undefined for $q = 7$ and $\emptyset \neq E \subseteq \{2, 7\}$ or for $q = 10$ and any $\emptyset \neq E \subseteq \{5, 10\}$. Similarly, $d_1(q, E, d)$ is undefined for $q = 1$ and $E = \{1\}$. This is so because $ds \in \mathcal{LT}(d_0, (1, \{1\}))$ and s is not controllable. The resulting automaton $\mathcal{A}_{d_1}$ is depicted in Figure 4.14.* ◇

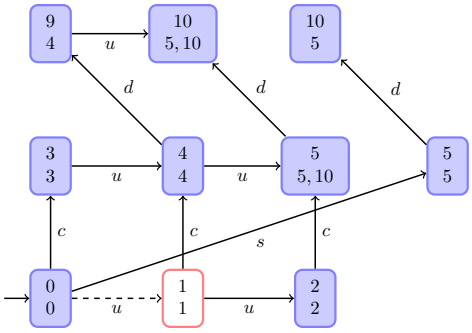


Figure 4.14: $\mathcal{A}_{d_1}$

Definition 4.6 *Let $d^\dagger = d_n$ for the least n such that $d_{n+1} = d_n$ where $d_{i+1} = \phi(d_i)$ and $d_0 : Q \times 2^Q \times \Sigma \to Q$ is the map defined with $d_0(q, E, \sigma) = \delta(q, \sigma)$.* •

Example 4.10 *Given that* $d_1(E, E, \Sigma_{ua}^*) = E \cup \{d_1(q, E, d) \mid q \in E\}$, $\mathcal{LE}(d_1)$ *contains exactly one loosing estimate, namely the singleton set* $\{1\}$. *Thus,* $u \in \mathcal{LT}(d_1, (0, \{0\}))$ *and* $d_2(0, \{0\}, u)$ *is undefined. The automaton* $\mathcal{A}_{d_2}$ *is depicted in Figure 4.15. The reader may verify that* $\phi(d_2) = d_2$.

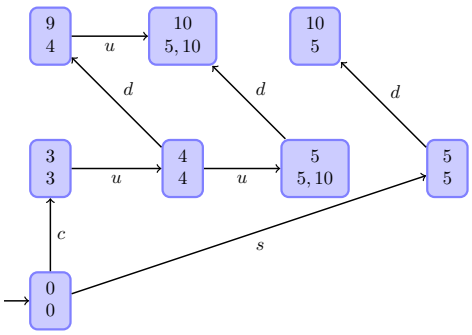


Figure 4.15: $\mathcal{A}_{d_2}$

Based on this definition, we can conclude that

Theorem 4.2 $K^{\uparrow} = \mathcal{L}(d^{\uparrow})$ *is the largest prefix-closed and controllable sub-language of* $\mathcal{L}(\mathcal{G})$ *s.t.* $\mathcal{L}_\varphi$ *is opaque w.r.t.* $K^{\uparrow}$ *and* Σ_a *and is regular.*

Example 4.11 *The optimal control defined by the automaton* $\mathcal{A}_{d_2}$ *prevents the agent from using the lift of the south wing, and it also prevents the agent from using the lift of the north wing from the second floor to the third floor at any time after he has used this lift downwards.* ◇

4.4.3 Conclusion

Given a system modeled by an $\mathcal{G}$ over Σ^*, a regular secret $\mathcal{L}_\varphi \subseteq \Sigma^*$ and an adversary observing a subset $\Sigma_a \subseteq \Sigma$ of the events of $\mathcal{G}$, we have addressed the problem of computing the supremal controller that enforces the opacity of $\mathcal{L}_\varphi$ on $\mathcal{G}$ while observing and controlling respective subsets Σ_m and $\Sigma_c \subseteq \Sigma_m$ of events of $\mathcal{G}$ under various assumptions on the set of events. The question is open whether the supremal controller is still regular and effectively computable when Σ_m and Σ_a are not comparable. We would like to add that the control synthesis algorithm which has been proposed can be easily adapted to enforce the notion of secrecy as defined in [AČZ06].

4.5 Ensuring opacity by dynamic filtering

In this section, instead of restricting the behavior of the system by means of a controller which enables/disables some actions, we consider *dynamic* observers that will dynamically change the set of observable events in order to ensure opacity. Compared to the approaches

related to the supervisory control theory, this approach is not intrusive in the sense that it does not restrict the system behavior but only hides observable events. Indeed, one can think of a dynamic observer as a monitor or a *filter* (See Figure 4.16) which is added on top of the system.

Figure 4.16: Architecture filtering out fequences of events in $\mathcal{G}$

We here show how to check opacity when the dynamic observer is given by a finite LTS, thus extending the results of the previous section. Second we give an algorithm to compute the set of all dynamic observers which can ensure opacity of a secret φ for a given system $\mathcal{G}$. Notice also that *secrecy* (see section 4.1) can be handled as a particular case of opacity and thus our framework applies to secrecy as well.

Remark 4.4 *The notion of* dynamic observers *was already introduced in [CT08] for the fault diagnosis problem. Notice that the fault diagnosis problem and the opacity problems are not reducible one to the other and thus we have to design new algorithms to solve the opacity problems under dynamic observations.*

4.5.1 Opacity with Dynamic Projection

So far, we have assumed that the observability of events is given a priori and this is why we used the term static projections/observers. We generalize this approach by considering the notion of *dynamic projections* encoded by means of *dynamic observers* as introduced in [CT08] for the fault diagnosis problem. In this section, we formulate the opacity problem using dynamic observers and introduce the notion of dynamic projection that permits to render unobservable some events after a given observed trace (for example, some outputs of the system). To illustrate the benefits of such projections, we consider the following example:

Example 4.12 *Consider again the LTS $\mathcal{G}$ of Example 4.2, Figure 4.3, where the set of secret states is $F = \{q_2, q_5\}$. With $\Sigma_o = \Sigma = \{a, b\}$, F is not opaque. If either $\Sigma_o = \{a\}$ or $\Sigma_o = \{b\}$, then the secret becomes opaque. Thus if we have to define static sets of observable events, at least one event will have to be permanently unobservable. However, the less you hide, the more important is the observable behavior of the system. Thus, we should try to reduce as much as possible the hiding of events. For this particular example, we can be more efficient by using a* dynamic *observer that will render unobservable an event only when necessary. In this example, after observing b^*, the attacker still knows that the system is in the initial state. However, if a subsequent "a" follows, then the attacker should not be able to observe "b" as in this case it could know that the system is in a secret state. We can then design a dynamic events hider as follows: at the beginning, every event is observable; when an "a" occurs, the observer hides any subsequent occurrences of "b" and permits only the observation of "a". Once an "a" has been observed, the observer releases its hiding by letting both "a" and "b" be observable again.*

4.5.1.1 Opacity Generalized to Dynamic Projection

In this section, we define the notion of dynamic projection and its associated dynamic observer. We show how to extend the notion of opacity in order to take into account the dynamic aspect of events' observability.

Dynamic projections and dynamic observers An (observation-based) dynamic projection is a function that decides to let an event be observable or to hide it (see Figure 4.16), thus playing the role of a filter between the system and the attacker to prevent information flow. Such a projection can be defined as follows:

Definition 4.7 *A* dynamic observability choice *is a mapping* $T_D : \Sigma^* \to 2^{\Sigma}$. *The (observation-based)* dynamic projection *induced by* T_D *is the mapping* $D : \Sigma^* \to \Sigma^*$ *defined by:* $D(\varepsilon) = \varepsilon$ *and for all* $u \in \Sigma^*$, *and all* $\sigma \in \Sigma$,

$$D(u.\sigma) = D(u).\sigma \text{ if } \sigma \in T_D(D(u)) \text{ and } D(u.\sigma) = D(u) \text{ otherwise.} \tag{4.2}$$

Assuming that $u \in \Sigma^*$ occurred in the system and $\mu \in \Sigma^*$ has been observed so far by the attacker i.e., $\mu = D(u)$, then the events that are currently observable are the ones which belong to $T_D(\mu)$. Note that the choice of this set cannot change until an observable event occurs in the system. Given $\mu \in \Sigma^*$, $D^{-1}(\mu) = \{u \in \Sigma^* \mid D(u) = \mu\}$ i.e., the set of sequences that project onto μ.

Example 4.13 *A dynamic projection* $D : \Sigma^* \to \Sigma^*$ *corresponding to the one we introduced in Example 4.12 can be induced by the dynamic observability choice* T_D *defined by* $\forall u \in b^*.a$, $T_D(u) = \{a\}$, *and* $T_D(u) = \{a, b\}$ *for all the other sequences* $u \in \Sigma^*$.

For a model $\mathcal{G}$ as above and a dynamic projection D, we denote by $Tr_D(\mathcal{G}) = D(\mathcal{L}(\mathcal{G}))$, the set of observed traces. Conversely, given $\mu \in Tr_D(\mathcal{G})$, the set of words $[\![\mu]\!]_D$ of $\mathcal{G}$ that are compatible with μ is defined by:

$$[\![\varepsilon]\!]_D = \{\varepsilon\} \quad \text{and for } \mu \in \Sigma^*,\ \sigma \in \Sigma : [\![\mu.\sigma]\!]_D = D^{-1}(\mu).\sigma \cap \mathcal{L}(\mathcal{G})$$

Given two different dynamic projections D_1 and D_2 and a system G over Σ, we say that D_1 and D_2 are G-equivalent, denoted $D_1 \sim_G D_2$, whenever for all $u \in \mathcal{L}(\mathcal{G})$, $D_1(u) = D_2(u)$. The relation $\sim_G$ identifies two dynamic projections when they agree on $\mathcal{L}(\mathcal{G})$; they can disagree on other words in Σ^* but since they will not be generated by $\mathcal{G}$, it will not make any difference from the attacker point of view. In the sequel we will be interested in computing the interesting part of dynamic projections given $\mathcal{G}$, and thus we will compute one dynamic projection in each class.

Definition 4.8 *Given an LTS* $\mathcal{G} = (Q, q_0, \Sigma, \longrightarrow)$, *with* $F \subseteq Q$, *the set of secret states,* F *is opaque with respect to* $(\mathcal{G}, D)$ *if*

$$\forall \mu \in Tr_D(\mathcal{G}),\ \mathrm{Post}_{\mathcal{G}}(\{q_0\}, [\![\mu]\!]_D) \not\subseteq F \tag{4.3}$$

Again, this definition extends to families of sets. We say that D is a valid dynamic projection if (4.3) is satisfied (i.e., whenever F is opaque w.r.t. (G, D)) and we denote by $\mathcal{D}$ the set of valid dynamic projections. Obviously if $D_1 \sim_G D_2$, then D_1 is valid if and only if D_2 is valid. We denote by $\mathcal{D}_{\sim_{\mathcal{G}}}$ the quotient set of $\mathcal{D}$ by $\sim_{\mathcal{G}}$.

Remark 4.5 *Let $\Sigma_o \subseteq \Sigma$ be a fixed subset of actions, then if D is a dynamic projection that defines a constant mapping making actions in Σ_o always observable (and the others always unobservable), we have $D(\mu) = P_{\Sigma_o}(\mu)$ and we retrieve the original definition of state based opacity in case of static projection. Finally, note that we can also alternatively consider a trace-based opacity as the one defined in Definition 4.1, with dynamic projection instead of natural projection with a result similar to the one of Proposition 4.2. The property of secrecy can be extended as well using dynamic projection.*

In the sequel, we will be interested in checking the opacity of F w.r.t. $(\mathcal{G}, D)$ or to synthesize such a dynamic projection D ensuring this property. In Section 4.1, the dynamic projection was merely the natural projection and computing the observational behavior of $\mathcal{G}$ was easy. Here, we need to find a characterization of these dynamic projections that can be used to check opacity or to enforce it. To do so, we introduce the notion of dynamic observer [CT08] that will encode a dynamic projection in terms of LTS's.

Definition 4.9 (Dynamic observer) *An* observer *is a complete and deterministic LTS $\mathcal{O} = (X, x_0, \Sigma, \delta_o, \Gamma)$ where X is a (possibly infinite) set of states, $x_0 \in X$ is the initial state, Σ is the set of input events, $\delta_o : X \times \Sigma \to X$ is the transition function (a total function), and $\Gamma : X \to 2^{\Sigma}$ is a labeling function that specifies the set of events that the observer keeps observable at state x. We require that for all $x \in X$ and for all $\sigma \in \Sigma$, if $\sigma \notin \Gamma(x)$, then $\delta_o(x, \sigma) = x$, i.e., if the observer does not want an event to be observed, it does not change its state when such an event occurs.*

We extend δ_o to words of Σ^* by: $\delta_o(q, \varepsilon) = q$ and for $u \in \Sigma^*, \sigma \in \Sigma$, $\delta_o(q, u.\sigma) = \delta_o(\delta_o(q, u), \sigma)$. Assuming that the observer is at state x and an event σ occurs, it outputs σ whenever $\sigma \in \Gamma(x)$ or nothing (ε) if $\sigma \notin \Gamma(x)$ and moves to state $\delta_o(x, \sigma)$. An observer can be interpreted as a functional transducer taking a string $u \in \Sigma^*$ as input, and producing the output which corresponds to the successive events it has chosen to keep observable.

Example 4.14 *Examples of dynamic observers are given in Figure 4.17.* ◇

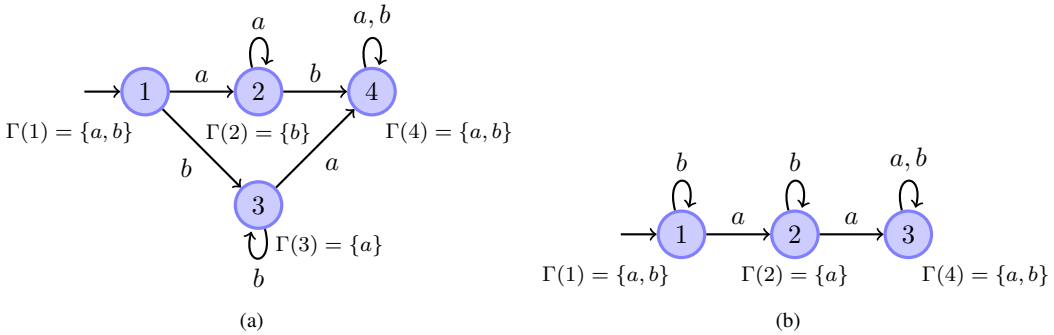


Figure 4.17: Examples of Dynamic Observers

We now relate the notion of dynamic observer to the notion of dynamic projection.

Proposition 4.8 *Let $\mathcal{O} = (X, x_0, \Sigma, \delta_o, \Gamma)$ be an observer and define $D_{\mathcal{O}}$ as follows: $D_{\mathcal{O}}(\varepsilon) = \varepsilon$, and for all $u \in \Sigma^*$, $D_{\mathcal{O}}(u.\sigma) = D_{\mathcal{O}}(u).\sigma$ if $\sigma \in \Gamma(\delta_o(x_0, u))$ and $D_{\mathcal{O}}(u)$ otherwise. Then $D_{\mathcal{O}}$ is a dynamic projection.*

In the sequel, we shall write $[\![\mu]\!]_{\mathcal{O}}$ for $[\![\mu]\!]_{D_{\mathcal{O}}}$.

Proposition 4.9 *Given a dynamic projection D and T_D its dynamic observability choice, we can define the dynamic observer $\mathcal{O}_D = (\Sigma^*, \varepsilon, \Sigma, \delta_D, T_D)$ where $\delta_D(u, \sigma) = D(u.\sigma)$.*

Note that there might exists several equivalent observers that encode the same dynamic projection. For example, the observer depicted in Figure 4.17(b) is one observer that encodes the dynamic projection described in Example 4.13. But, one can consider other observers obtained by unfolding an arbitrary number of times the self-loops in states 1 or 3. Finally, to mimic the language theory terminology, we will say that a dynamic projection D is *regular* whenever there exists a finite state dynamic observer $\mathcal{O}$ such that $D_{\mathcal{O}} = D$.

To summarize this part, we can state that with each dynamic projection D, we can associate a dynamic observer $\mathcal{O}_D$ such that $D = D_{\mathcal{O}_D}$. In other words, we can consider a dynamic projection or one of its associated dynamic observers whenever one representation is more convenient than the other. If the dynamic projection D derived from $\mathcal{O}$ is valid, we say that $\mathcal{O}$ is a *valid* dynamic observer. In that case, we will say that F is opaque w.r.t. $(\mathcal{G}, \mathcal{O})$ and we denote by $\mathcal{OBS}(\mathcal{G})$ the set of all valid dynamic observers.

Checking opacity The first problem we are going to address consists in checking whether a given dynamic projection ensures opacity. To do so, we assume given a dynamic observer which defines this projection map. The problem, we are interested in, is then the following:

Problem 4.2 (Dynamic State Based Opacity Problem)

INPUT*:* *A non-deterministic LTS $\mathcal{G} = (Q, q_0, \Sigma, \delta, F)$ and a dynamic observer $\mathcal{O} = (X, x_0, \Sigma, \delta_o, \Gamma)$.*

PROBLEM*:* *Is F opaque w.r.t. $(\mathcal{G}, \mathcal{O})$?*

We first construct an LTS which represents what an attacker will see under the dynamic choices of observable events made by $\mathcal{O}$ (i.e. by hiding in $\mathcal{G}$ the events the observer has chosen to hide after observing a given trace). To do so, we define the LTS $\mathcal{G} \otimes \mathcal{O} = (Q \times X, (q_0, x_0), \Sigma \cup \{\tau\}, \delta, F \times X)$ where τ is a fresh letter not in Σ and δ is defined for each $\sigma \in \Sigma$, and $(q, x) \in Q \times X$ by:

- $\delta((q, x), \sigma) = \delta_{\mathcal{G}}(q, \sigma) \times \{\delta_o(x, \sigma)\}$ if $\sigma \in \Gamma(x)$;
- $\delta((q, x), \tau) = \big(\cup_{\sigma \in \Sigma \setminus \Gamma(x)} \delta_{\mathcal{G}}(q, \sigma)\big) \times \{x\}$.

Proposition 4.10 *F is opaque w.r.t. $(\mathcal{G}, \mathcal{O})$ if and only if $F \times X$ is opaque w.r.t. to $(\mathcal{G} \otimes \mathcal{O}, \Sigma)$.*

The previous result is general, and if $\mathcal{O}$ is a finite state observer we obtain the following theorem:

Theorem 4.3 *For finite state observers, Problem 4.2 is PSPACE-complete.*

As Proposition 4.10 reduces the problem of checking opacity with dynamic observers to the problem of checking opacity with static observers, Theorem 4.3 extends to family of sets (and thus to secrecy).

4.5.2 Enforcing opacity with dynamic projections

So far, we have assumed that the dynamic projection/observer was given. Next we will be interested in *synthesizing* one in such a way that the secret becomes opaque w.r.t. the system and this observer. Initially we assume that the attacker can observe all events in Σ. The problem we have in mind is to add an interface (a dynamic observer/projection) between the system and the attacker that will filter out some events so that the confidential information never leaks, following the architecture of Figure 4.16. Thus the problem can be stated as follows:

Problem 4.3 (Dynamic Observer Synthesis Problem)
INPUT*:* *A non-deterministic LTS* $\mathcal{G} = (Q, q_0, \Sigma, \delta, F)$.
PROBLEM*:* *Compute the set of valid observers* $\mathcal{OBS}(\mathcal{G})$[3].

Deciding the existence of a valid observer is trivial: it is sufficient to check whether always hiding Σ is a solution. Moreover, note that $\mathcal{OBS}(\mathcal{G})$ can be infinite and that there might be an infinite number of different valid projections/observers ensuring the opacity of F with respect to $\mathcal{G}$.

To solve Problem 4.3, we reduce it to a safety 2-player game. Player 1 will play the role of an observer and Player 2 what the attacker observes. Assume the LTS $\mathcal{G}$ can be in any of the states $s = \{q_1, q_2, \cdots, q_n\}$, after a sequence of actions occurred. A round of the game is: given s, Player 1 chooses which letters should be observable next i.e., a set $t \subseteq \Sigma$; then it hands it over to Player 2 who picks up an observable letter $\sigma \in t$; this determines a new set of states on which $\mathcal{G}$ can be after observing σ, and the turn is back to Player 1. The goal of the Players are defined by:

- The goal of Player 2 is to pick up a sequence of letters such that the set of states that can be reached after this sequence is included in F. If Player 2 can do this, then it can infer the secret F. Player 2 thus plays a *reachability game* trying to enforce a particular set of states, say Bad (i.e., the states in which the secret is disclosed).
- The goal of Player 1 is opposite: it must keep the game in a safe set of states where the secret is not disclosed. Thus Player 1 plays a *safety game* trying to keep the game in the complement set of Bad.

As we are playing a (finite) turn-based game, Player 2 has a strategy to enforce Bad iff Player 1 has no strategy to keep the game in the complement set of Bad (turn-based finite games are *determined* [Mar75]).
We now formally defines the 2-player game and show that it allows us to obtain a finite representation of all the valid dynamic observers. Let $H = (S_1 \cup S_2, s_0, M_1 \cup M_2, \delta_H)$ be a deterministic game LTS given by:

- $S_1 = 2^Q$ is the set of Player 1 states and $S_2 = 2^Q \times 2^\Sigma$ the set of Player 2 states;
- the initial state of the game is the Player 1 state $s_0 = \{q_0\}$;
- Player 1 will choose a set of events to hide in Σ. Thus, Player 1 actions are in the alphabet $M_1 = 2^\Sigma$ and Player 2 actions in $M_2 = \Sigma$;
- the transition relation $\delta_H \subseteq (S_1 \times M_1 \times S_2) \cup (S_2 \times M_2 \times S_1)$ is given by:

- Player 1 moves (choice of events to observe): if $s \in S_1, t \subseteq \Sigma$, then $\delta_H(s,t) = (s,t)$;
- Player 2 moves (choice of next observable event): if $(s,t) \in S_2$, $\sigma \in t$ and $s' = \mathrm{Post}(s, (\Sigma \setminus t)^*.\sigma) \neq \emptyset$, then $\delta_H((s,t),\sigma) = s'$.

We define the set of Bad states to be the set of Player 1 states s s.t. $s \subseteq F$. For family of sets $F_1, F_2, \cdots, F_k$, Bad is the set of states $2^{F_1} \cup 2^{F_2} \cup \cdots \cup 2^{F_k}$.

Remark 4.6 *If we want to exclude the possibility of hiding everything for Player 1, it suffices to build the game H with this constraint on Player 1 moves (i.e., $\forall s \in S_1$, $\mathsf{enabled}(s) \neq \emptyset$). Using a similar method, we can also consider other kinds of constraints: for example, a valid observer could choose to hide outputs whenever it is necessary to preserve the secret, however, this observer must keep observable (and thus accepted) all the requests sent by the attacker to the system. To do so, assuming that Σ is partitioned into $\Sigma_? \cup \Sigma_!$, where $\Sigma_?$ denotes the set of inputs of the system and $\Sigma_!$ the set of outputs events, we can force the observer to choose to hide only events of $\Sigma_!$, letting observable all the actions performed by the attacker by building the game H so that $\forall s \in S_1$, $2^{\Sigma_?} \subseteq \mathsf{enabled}(s)$.* □

Let $\mathit{Runs}_i(H), i = 1, 2$ be the set of runs of H that end in a Player i state. A *strategy* for Player i is a mapping $f_i : \mathit{Runs}_i(H) \to M_i$ that associates with each run that ends in a Player i state, the new choice of Player i. Given two strategies f_1, f_2, the game H generates the set of runs $\mathit{Outcome}(f_1, f_2, H)$ combining the choices of Players 1 and 2 w.r.t. f_1 and f_2. f_1 is a *winning strategy* for Playing 1 in H for avoiding Bad if for all Player 2 strategies f_2, no run of $\mathit{Outcome}(f_1, f_2, H)$ contains a Bad state. A winning strategy for Player 2 is a strategy f_2 s.t. for any strategy f_1 of Player 1, $\mathit{Outcome}(f_1, f_2, H)$ reaches a Bad state. As turn-based games are determined, either Player 1 has a winning strategy or Player 2 has a winning strategy.

We now relate the set of winning strategies for Player 1 in H to the set of valid dynamic projections. Let $P_{M_2}(\varrho) = P_\Sigma(\mathit{tr}(\varrho))$ for a run ϱ of H.

Definition 4.10 *Given a dynamic projection D, we define a strategy f_D such that for every $\varrho \in \mathit{Runs}_1(H)$, $f_D(\varrho) = T_D(P_{M_2}(\varrho))$.* ■

Let $\mathit{Outcome}_1(f_1, H) = (\cup_{f_2} \mathit{Outcome}(f_1, f_2, H)) \cap \mathit{Runs}_1(H)$ be the set of runs ending in a Player 1 state which can be generated in the game when Player 1 plays f_1 against all the possible strategies of Player 2. The set of runs $\mathit{Outcome}_2(f_2, H)$ is similarly defined.

Proposition 4.11 *Let D be a dynamic projection. D is valid if and only if f_D is a winning strategy for Player 1 in H.*

Given a strategy f for Player 1 in H, for all $\mu \in \Sigma^*$, there exists at most one $\varrho_\mu \in \mathit{Outcome}_1(f, H)$ such that $P_{M_2}(\mathit{tr}(\varrho_\mu)) = \mu$.

Definition 4.11 *Let f be a strategy for Player 1 in H. We define the dynamic projection D_f induced by the dynamic observability choice $T_f : \Sigma^* \to 2^\Sigma$ given by: $T_f(\mu) = f(\varrho_\mu)$ if ϱ_μ.* ■

Proposition 4.12 *If f is a winning strategy for Player 1 in H, then D_f is a valid dynamic observer.*

Notice that we only generate a representative for each of the equivalence classes induced by $\sim_{\mathcal{G}}$. However, an immediate consequence of the two previous propositions is that there is a bijection between the set of winning strategies of Player 1 and $\mathcal{D}_{\sim_{\mathcal{G}}}$.

4.5.2.1 Most Permissive Dynamic Observer

We now define the notion of *most permissive* dynamic observers and show the existence of a most permissive dynamic observer for a system $\mathcal{G}$. For an observer $\mathcal{O} = (X, x_o, \Sigma_o, \delta_o, \Gamma)$ and $s \in \Sigma^*$, recall that $\Gamma(\delta_o(x_o, s))$ is the set of events that $\mathcal{O}$ chooses as observable after observing s. Assume $s = \sigma_0.\sigma_1 \cdots \sigma_k$. Let $\overline{s} = \Gamma(x_o).\sigma_1.\Gamma(\delta_o(x_o, \sigma_1)).\sigma_2.\Gamma(\delta_o(x_o, \sigma_2)) \cdots \sigma_k.\Gamma(\delta_o(x_o, \sigma_k)))$ i.e., $\overline{s}$ contains the history of what $\mathcal{O}$ has chosen to observe at each step and the events that occurred after each choice.

Definition 4.12 *Let $\mathcal{O}^* : (2^{\Sigma}.\Sigma)^* \to 2^{2^{\Sigma}}$. By definition, such a mapping $\mathcal{O}^*$ is the most permissive observer*[4] *for ensuring that F is opaque if the following holds:*

$$\mathcal{O} = (X, x_o, \Sigma_o, \delta_o, \Gamma) \text{ is a valid observer} \iff \forall w \in \mathcal{L}(\mathcal{G}),\ \Gamma(\delta_o(x_o, w)) \in \mathcal{O}^*(\overline{w})$$

The definition of the most permissive observer states that any valid observer $\mathcal{O}$ must choose a set of observable events in $\mathcal{O}^*(\overline{w})$ on input w; if an observer chooses its set of observable events in $\mathcal{O}^*(\overline{w})$ on input w, then it is a valid observer.

Theorem 4.4 *The most permissive dynamic observer $\mathcal{O}^*$ can be represented by a finite LTS $\mathcal{F}_H$.*

The previous theorem states that $\mathcal{F}_H$ can be used to generate any observer. In particular, given a state-based winning strategy, the corresponding valid observer is finite and thus its associated dynamic projection is regular.

An immediate corollary of Theorem 4.4 is the following:

Corollary 4.1 *Problem 4.3 is in EXPTIME.*

Example 4.15 *To illustrate this section, we consider the following small example. The system is depicted by the LTS in Figure 4.18(a). The set of secret states is reduced to the state (2). Figure 4.18(b) represents the associated game LTS. The states of Player 1 are represented by circles whereas the ones of Player 2 are represented by squares. The only bad states is the state (2) (bottom right). The most permissive strategy is obtained when Player 1 does not allow transition $\{a, b\}$ to be triggered in state (1) (otherwise, Player 2 could chose to observe either event a or b and in this case the game will evolve into state (2) and the secret will be revealed). The dashed lines represent the transitions that are removed from the game LTS to obtain $\mathcal{F}_H$. Finally, Figure 4.18(c) represents a possible observer $\mathcal{O}$ generated from the game LTS.* ◇

[4] Strictly speaking $\mathcal{O}^*$ is not an observer because it maps to sets of sets of events whereas observers map to sets of events. Still we use this term because it is the usual terminology in the literature.

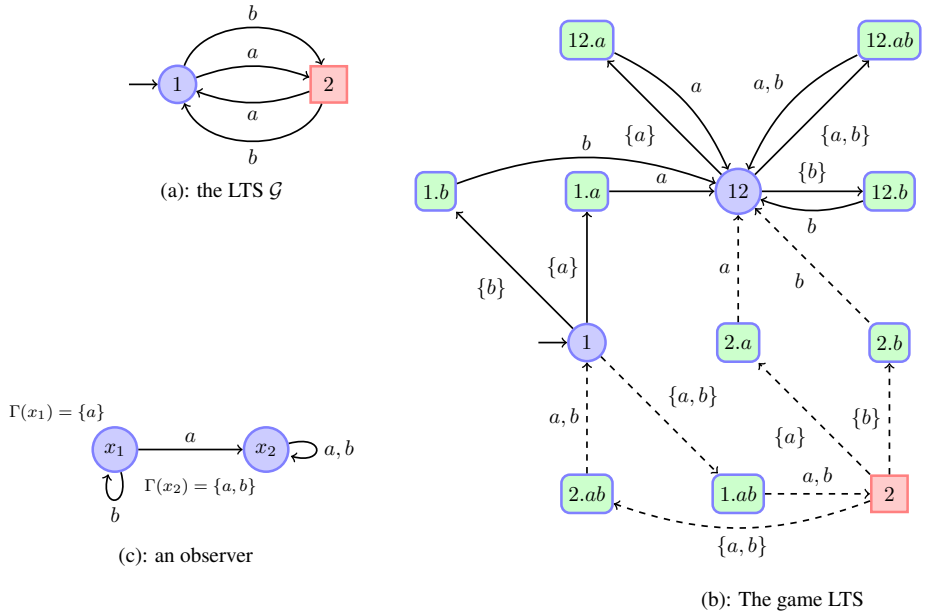


Figure 4.18: Example of a game LTS

4.5.2.2 Optimal Dynamic Observer

Among all the possible observers that ensure the opacity of the secret, it is worthwhile noticing that some are better (in some sense) than others: they hide less events on *average*. We here define a notion of cost for observers which captures this intuitive notion. We first introduce a general cost function and we show how to compute the cost of a given pair $(\mathcal{G}, \mathcal{O})$ where $\mathcal{G}$ is a system and $\mathcal{O}$ a finite state observer. Second, we show that among all the valid observers (that ensure opacity), there is an optimal cost, and we can compute an observer which ensures this cost. The problems in this section and the solutions are closely related to the results in [CT08] and use the same tools: Karp's mean-weight algorithm [Kar78] and a result of Zwick and Paterson [ZP96]. We want to define a notion of cost which takes into account the set of events the observer chooses to hide and also how long it hides them. We assume that the observer is a finite LTS $\mathcal{O} = (X, x_0, \Sigma, \delta_o, \Gamma)$. With each set of observable events $\Sigma' \in 2^\Sigma$ we associate a *cost of hiding* $\Sigma \setminus \Sigma'$ which is a positive integer. We denote $\mathit{Cost} : 2^\Sigma \to \mathbb{N}$ this function. Now, if $\mathcal{O}$ is in state x, the current cost per time unit is $\mathit{Cost}(\Gamma(x))$. Let $\mathit{Runs}^n(\mathcal{G})$ be the set of runs of length n in $\mathit{Runs}(\mathcal{G})$. Given a run $\rho = q_0 \xrightarrow{\sigma_1} q_1 \cdots q_{n-1} \xrightarrow{\sigma_n} q_n \in \mathit{Runs}^n(\mathcal{G})$, let $x_i = \delta_o(x_0, w_i)$ with $w_i = \mathit{tr}(\rho[i])$. The *cost* associated with $\rho \in \mathit{Runs}^n(\mathcal{G})$ is defined by:

$$\mathit{Cost}(\rho, \mathcal{G}, \mathcal{O}) = \frac{1}{n+1} \cdot \sum_{i=0..n} \mathit{Cost}(\Gamma(x_i)).$$

Notice that the time basis we take is the number of steps which occurred in $\mathcal{G}$. Thus if

the observer is in state x, and chooses to observe $\Gamma(x)$ at steps i and $i+1$, *Cost*$(\Gamma(x))$ will be counted twice: at steps i and $i+1$. The definition of the cost of a run corresponds to the average cost per time unit, the time unit being the number of steps of the run in $\mathcal{G}$. Define the cost of the set of runs of length n that belongs to $Runs^n(\mathcal{G})$ by: *Cost*$(n, \mathcal{G}, \mathcal{O}) = \max\{$*Cost*$(\rho, \mathcal{G}, \mathcal{O}) \mid \rho \in Runs^n(\mathcal{G})\}$. The *cost of an observer* with respect to a system $\mathcal{G}$ is

$$\textit{Cost}(\mathcal{G}, \mathcal{O}) = \limsup_{n \to \infty} \textit{Cost}(n, \mathcal{G}, \mathcal{O}) \tag{4.4}$$

(notice that the limit may not exist whereas the limit sup is always defined.) To compute the cost of a given observer, we can use a similar algorithm as the one given in [CT08], and using Karps's maximum mean-weight cycle algorithm [Kar78]:

Theorem 4.5 *Computing Cost*$(\mathcal{G}, \mathcal{O})$ *is in PTIME.*

Finally we can solve the following optimization problem:

Problem 4.4 (Bounded Cost Observer)
INPUTS: *A LTS* $\mathcal{G} = (Q, q_0, \Sigma, \delta, F)$ *and an integer* $k \in \mathbb{N}$.
PROBLEMS:

(A) Is there any $\mathcal{O} \in \mathcal{OBS}(\mathcal{G})$ *s.t.* F *is opaque w.r.t.* $(\mathcal{G}, \mathcal{O})$ *and Cost*$(\mathcal{G}, \mathcal{O}) \leq k$ *?*

(B) If the answer to (A) is "yes", compute a witness observer $\mathcal{O}$ *s.t. Cost*$(\mathcal{G}, \mathcal{O}) \leq k$.

To solve this problem we use a result from Zwick and Paterson [ZP96], which is an extension of Karp's algorithm for finite state games.

Theorem 4.6 *Problem 4.4 can be solved in EXPTIME.*

The solution to this problem is the same as the one given in [CT08], and the proof for the opacity problem is detailed in [J9]. The key result is Theorem 4.4, which enables us to represent all the winning strategies in H as a finite LTS. Synchronizing $\mathcal{G}$ and the most permissive valid dynamic observer $\mathcal{F}_H$ produces a *weighted game*, the optimal value of which can be computed in PTIME (in the size of the product) using the algorithm in [ZP96]. The optimal strategies can be computed in PTIME as well. As $\mathcal{G} \times \mathcal{F}_H$ has size exponential in $\mathcal{G}$ and Σ, the result follows.

4.6 General Conclusion and Futur Work

In this chapter, we have investigated different kinds of problems related to the notion of opacity : the verification of opacity, the diagnosis of information leakage as well as the control of opacity by considering the supervisory control theory and the control by means of dynamic observer.

The work presented in this chapter suggests several other problems that could be interesting to investigate.

- So far, we have assumed that the attacker is passive in the sense that he/she is observing the system and was deducing information according to he observation of the system. It might be interesting to consider an active attacker that might be able to dynamically observe more events (with a price to pay), to change the decision of the controller by allowing to disable events to be triggered by the system, etc. A first step toward this last point has been proposed in [KWKL16] and deserves to be continued and extended.

- In section 4.4, we provided a solution to the opacity control problem with the assumption that the set of events observable by the controller and the one observable by the attacker are comparable. In order to provide a complete theory for the control problem, it would be interesting to investigate, how to remove the assumptions that relates the alphabets. It is still an open problem, even-though we suspect this problem to be undecidable. Note that, in [TML$^+$16], the authors remove these assumptions, but they assume that the attacker does not know the controller (or in other words, does not know the set of events observed by the controller). Therefore, in this setting, removing the set of sequences that reveal the secret is sufficient to solve the problem (no new sequences revealing the secret can appear after the first step of control).

- So far, we have been interested in the analysis of the opacity of a system w.r.t. to a single attacker. In [BBB$^+$07], the authors consider several attackers having a different point of view of the system, each of them trying to infer a particular secret. They investigate the control problem with the hypothesis that the controller observes and controls all events. Applying our techniques of control to their framework can thus be an interesting extension. Also one can think that each attacker has its own set of secrets and that they want to infer the secrets of the other attackers. On possible idea would be to build a "fair" controller that would ensure that each attacker can only acquire a minimal knowledge of the others secrets. This can be seen as a game with costs.

- Many applications in which security issues cannot be ignored deal with infinite data types. Such systems or services are naturally modeled with infinite transition systems. In order to avoid that confidential information leaks from such infinite systems, it seems important to investigate combined techniques of opacity control and abstract interpretation. A possible scheme is to enforce opacity by supervisory control on finite abstractions of infinite systems and then to lift control to the original systems for subsequent model-checking.

- Finally, an interesting extension of our work would be to consider quantitative opacity rather than qualitative opacity. Indeed it might happen than the set of trajectories that violate the opacity property has a very low (resp. high) probability to be executed. In the first case, we can consider that a controller or a diagnoser can disregard this set of sequences in order to take its decision. In the second case, it is the attacker that can assume that the secret is leaked as soon as the possible sequences based on its observation have a probability to be executed that exeeds a given trehshold. This problem has been tackled in e.g., [BMS10], where the authors try to quantify the

probability of opacity leakage and to minimize it by means of control. One might try to extend the techniques introduced in Section 4.4 to this new setting. Another direction for quantitative opacity, would be to quantify the effort needed by the attacker to deduce the secret and to declare that a system is secure w.r.t. a given secret whenever the effort to acquire these secret is too important.

Chapter 5

Conclusion & Perspectives

In this document, we have been interested in the analysis of discrete event systems. These analysis were performed on monolithic or concurrent systems. In all cases, the model of the system was fixed as well as the properties to be either ensured and / or diagnoses.

However, the last past years have seen the proliferation of cloud computing and of the Softwarization of Everything which tends to gather under the same roof computing (clouds/fogs), networking (SDN) and complex networked services oriented to the user (apps, IoT). For these applications, the growing scale together with the continuously changing runtime environments and user needs is significantly increasing their operational costs. In this domain, we would like to move from static towards adaptive controller synthesis solutions. The inherent dynamicity of deployed systems and the difficulty to forecast their evolution made it necessary for applications to dynamically adapt themselves at runtime with minimal or no human intervention. However, as changes to the system are performed autonomously, it is necessary to ensure that performed actions will not jeopardize the safety and the security of the running applications as well as the integrity and confidentiality the human using these devices.

The objective would be to promote a systematic and automatic approach for the design and implementation of reliable adaptive and reconfigurable systems and to provide tools ensuring that, whatever the runtime conditions, whatever the adaptations performed the system remains in safe configurations at a low cost. Such systems can be modeled with finite automata and their extensions. In this context, requirement/properties that have to be guaranteed are also dynamic and depend on the system configuration. These properties can be either logical or quantitative. One important aspect concerns the dynamicity of the systems in which components can arrive and leave at runtime. This leads to consider different requirements that have to be ensured on the global system depending on its current configuration. On-line synthesis techniques have thus to be designed to automatically deploy supervisors or controllers in charge of the safety of this system. On one side, an important challenge for this kind of reconfigurable system concerns the modular control that consists in decomposing global properties into several smaller ones that depend on subsystems or concern fewer attributes. To achieve the global adaptation strategy, those controllers have first to be computed and coordinated at a higher level. Finally, one has to improve the flex-

ibility of these controllers, allowing them to take potential failures and their diagnosis into account. A second step will be to consider the modeling of systems for which some part of the behavior is not known on-line. The goal will be to understand how such modeling can be articulated with the use of a constructive formal method like controller synthesis. In particular, how one can compute on-line controllers ensuring some dynamic properties that depend on the current configuration of the system.

Chapter 6

Personal Bibliography

This list of publications corresponds to the articles that are referenced in this document. A complete list can be found at the following address:
http://www.irisa.fr/sumo/Publis/Auteur/Herve.Marchand.english.html

Academic Journals

[J1] Y. Falcone, T. Jéron, H. Marchand, S. Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Science of Computer Programming*, 123:2-41, 2016.

[J2] Y. Falcone, H. Marchand. Enforcement and Validation (at runtime) of Various Notions of Opacity. *Discrete Event Dynamic Systems : Theory and Applications*, 25(4): 531-570, 2015.

[J3] P. Darondeau, H. Marchand, L. Ricker. Enforcing Opacity of Regular Predicates on Modal Transition Systems. *Discrete Event Dynamic Systems : Theory and Applications*, 25(1):251-270, 2015.

[J4] S. Chédor, Ch. Morvan, S. Pinchinat, H. Marchand. Enforcement and Validation (at runtime) of Various Notions of Opacity. *Discrete Event Dynamic Systems : Theory and Applications*, 25(1):271-294, 2015.

[J5] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, O. Nguema Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381-422, 2014.

[J6] G. Kalyon, T. Le Gall, H. Marchand, T. Massart. Symbolic Supervisory Control of Distributed Systems with Communications. *IEEE Transaction on Automatic Control*, 59(2):396-408, February 2014.

[J7] L. Hélouet, H. Marchand, B. Genest, T. Gazagnaire. Diagnosis from Scenarios, and applications. *Discrete Event Dynamic Systems : Theory and Applications*, 2013.

[J8] G Delaval, E. Rutten, H. Marchand. Integrating Discrete Controller Synthesis in a Reactive Programming Language Compiler. *Discrete Event Dynamic Systems : Theory and Applications*, 23(4):385-418, December 2013.

[J9] F. Cassez, J. Dubreil, H. Marchand. Synthesis of Opaque Systems with Static and Dynamic Masks. *Formal Methods in System Design*,, 40(1):88-115, 2012.

[J10] G. Kalyon, T. Le Gall, H. Marchand, T. Massart. Symbolic Supervisory Control of Infinite Transition Systems under Partial Observation using Abstract Interpretation. *Discrete Event Dynamic Systems : Theory and Applications*, 22(2):121-161, 2012.

[J11] G. Kalyon, T. Le Gall, H. Marchand, T. Massart. Decentralized Control of Infinite Systems. *Discrete Event Dynamic Systems : Theory and Applications*, 21(3):359-393, September 2011.

[J12] J. Dubreil, P. Darondeau, H. Marchand. Supervisory Control for Opacity. *IEEE Transactions on Automatic Control*, 55(5):1089-1100, May 2010.

[J13] E. Rutten, H. Marchand. Automatic generation of safe handlers for multi-task systems. *Journal of Embedded Computing*, 3(4):255-276, 2009.

[J14] B. Gaudin, H. Marchand. An Efficient Modular Method for the Control of Concurrent Discrete Event Systems: A Language-Based Approach. *Discrete Event Dynamic System*, 17(2):179-209, 2007.

[J15] C. Constant, T. Jéron, H. Marchand, V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558-574, August 2007.

[J16] B. Gaudin, H Marchand. Supervisory Control of Product and Hierarchical Discrete Event Systems. *European Journal of Control*, Special issue of European Control Conference, ECC 2003, 10(2), 2004.

[J17] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan. Formal Verification of programs specified with SIGNAL : Application to a Power Transformer Station Controller. *Science of Computer Programming*, 41(1):85-104, August 2001.

[J18] H. Marchand, P. Bournai, M. Le Borgne, P. Le Guernic. Synthesis of Discrete-Event Controllers based on the Signal Environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325-346, October 2000.

[J19] H. Marchand, M. Samaan. Incremental Design of a Power Transformer Station Controller using Controller Synthesis Methodology. *IEEE Transaction on Software Engineering*, 26(8):729-741, August 2000.

International Conferences

[C1] E. Fabre, L. Hélouet, E. Lefaucheux, H. Marchand. Diagnosability of Repairable Faults. In *13th International Workshop on Discrete Event Systems*, Pages 256-262, Xi'an, China, 2016.

[C2] N. Berthier, H. Marchand. Deadlock-free Discrete Controller Synthesis for Infinite State Systems. In 54th IEEE Conference on Decision and Control, Pages 1000-1007, Osaka, Japan, December 2015.

[C3] M. Renard, Y. Falcone, A. Rollet, S. Pinisetty, T. Jéron, H. Marchand. Enforcement of (Timed) Properties with Uncontrollable Events. In 12th International Colloquium on Theoretical Aspects of Computing (ICTAC 2015), Theoretical Aspects of Computing - ICTAC 2015, Volume LNCS, Cali, Colombia, October 2015.

[C4] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand. TiPEX: A Tool Chain for Timed Property Enforcement During eXecution. In RV'2015, 6th International Conference on Runtime Verification, Ezio Bartocci, Rupak Majumdar (eds.), Lecture Notes in Computer Science, Volume 9333, Vienne, Austria, September 2015.

[C5] N. Berthier, X. An, H. Marchand. Towards Applying Logico-numerical Control to Dynamically Partially Reconfigurable Architectures. In 5th IFAC International Workshop On Dependable Control of Discrete Systems - DCDS'15, Volume 48, Pages 132-138, Cancun, Mexico, May 2015.

[C6] X. An, G. Delaval, J-P. Diguet, A. Gamatie, A. Gueye, H. Marchand, N. De Palma, E. Rutten. Discrete Control-Based Design of Adaptive and Autonomic Computing Systems. In 11th International Conference on Distributed Computing and Internet Technology, ICDCIT 2015, Volume LNCS, Bhubaneswar, India, February 2015.

[C7] N. Berthier, H. Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. In *IEEE International Workshop on Discrete Event Systems*, Pages 46-53, Cachan, France, May 2014.

[C8] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand. Runtime Enforcement of Parametric Timed Properties with Practical Applications. In *IEEE International Workshop on Discrete Event Systems*, Pages 420-427, Cachan, France, May 2014.

[C9] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand. Runtime Enforcement of Regular Timed Properties. In *Software Verification and Testing, track of the Symposium on Applied Computing ACM-SAC 2014*, Pages 1279-1286, Gyeongju, Korea, March 2014.

[C10] Y. Falcone, H. Marchand. Runtime Enforcement of K-step Opacity. In *52nd IEEE Conference on Decision and Control*, Pages 7271-7278, Florence, Italy, December 2013.

[C11] S. Chédor, C. Morvan, S. Pinchinat, H. Marchand. Analysis of partially observed recursive tile systems. In *11th Int. Workshop on Discrete Event Systems*, Pages 265-271, Guadalajara, Mexico, October 2012.

[C12] G. Kalyon, T. Le Gall, H. Marchand, T. Massart. Synthesis of Communicating Controllers for Distributed Systems. In *50th IEEE Conference on Decision and Control and European Control Conference*, pp. 1803-1810, Orlando, USA, December 2011.

[C13] G. Kalyon, T. Le Gall, H. Marchand, T. Massart. Global State Estimates for Distributed Systems. In *31th IFIP International Conference on FORmal TEchniques for Networked and Distributed Systems, FORTE*, LNCS, Vol 6722, pp. 198-212, Reykjavik, Iceland, June 2011.

[C14] P. Darondeau, J. Dubreil, H. Marchand. Supervisory Control for Modal Specifications of Services. In *Workshop on Discrete Event Systems, WODES'10*, pp. 428-435, Berlin, Germany, August 2010.

[C15] E. Dumitrescu, A. Girault, H. Marchand, E. Rutten. Multicriteria optimal discrete controller synthesis for fault-tolerant real-time tasks. In *Workshop on Discrete Event Systems, WODES'10*, pp. 366-373, Berlin, Germany, August 2010.

[C16] H. Marchand, J. Dubreil, T. Jéron. Automatic Testing of Access Control for Security Properties. In *TestCom'09*, LNCS, Vol 5826, pp. 113-128, November 2009.

[C17] F. Cassez, J. Dubreil, H. Marchand. Dynamic Observers for the Synthesis of Opaque Systems. In *7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, Z. Liu, A.P. Ravn (Eds), LNCS, Vol 5799, pp. 352-367, Macao SAR, China, October 2009.

[C18] J. Dubreil, T. Jéron, H. Marchand. Monitoring Confidentiality by Diagnosis Techniques. In *European Control Conference*, pp. 2584-2590, Budapest, Hungary, August 2009.

[C19] T. Jéron, H. Marchand, S. Genc, S. Lafortune. Predictability of Sequence Patterns in Discrete Event Systems. In *IFAC World Congress*, pp. 537-453, Seoul, Korea, July 2008.

[C20] J. Dubreil, P. Darondeau, H. Marchand. Opacity Enforcing Control Synthesis. In *Workshop on Discrete Event Systems, WODES'08*, pp. 28-35, Gothenburg, Sweden, March 2008.

[C21] E. Dumitrescu, A. Girault, H. Marchand, E. Rutten. Optimal discrete controller synthesis for the modeling of fault-tolerant distributed systems. In *First IFAC Workshop on Dependable Control of Discrete Systems (DCDS'07)*, Paris, France, June 2007.

[C22] T. Jéron, H. Marchand, S. Pinchinat, M-O. Cordier. Supervision Patterns in Discrete Event Systems Diagnosis. In *Workshop on Discrete Event Systems, WODES'06*, Also published in DX'06, Penaranda de Duero (Burgos, Spain), pp. 262-268, Ann-Arbor (MI, USA), July 2006.

[C23] K. Schmidt, H. Marchand, B. Gaudin. Modular and Decentralized Supervisory Control of Concurrent Discrete Event Systems Using Reduced System Models. In *Workshop on Discrete Event Systems, WODES'06*, pp. 149-154, Ann-Arbor (MI, USA), July 2006.

[C24] B. Gaudin, H. Marchand. Supervisory Control and Deadlock Avoidance Control Problem for Concurrent Discrete Event Systems. In *44nd IEEE Conference on Decision and Control (CDC'05) and Control and European Control Conference ECC 2005*, pp. 2763-2768, Seville (Spain), December 2005.

[C25] B. Gaudin, H. Marchand. Efficient Computation of supervisors for loosely synchronous Discrete Event Systems: A State-Based Approach. In *6th IFAC World Congress*, Prague, Czech Republic, July 2005.

[C26] V. Rusu, H. Marchand, T. Jéron. Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems. In *Formal Methods 2005 (FM05)*, John Fitzgerald, Andrzej Tarlecki, Ian Hayes (Eds), LNCS, Vol 3582, pp. 189-204, July 2005.

[C27] B. Gaudin, H. Marchand. Safety Control of Hierarchical Synchronous Discrete Event Systems: A State-Based Approach. In *13th Mediterranean Conference on Control and Automation*, pp. 889-895, Limassol, Cyprus, June 2005.

[C28] B. Gaudin, H. Marchand. Modular Supervisory Control of a class of Concurrent Discrete Event Systems. In *Workshop on Discrete Event Systems, WODES'04*, pp. 181-186, September 2004.

[C29] H. Marchand, B. Gaudin. Supervisory Control Problems of Hierarchical Finite State Machines. In *41th IEEE Conference on Decision and Control*, pp. 1199-1204, Las Vegas, USA, December 2002.

[C30] H. Marchand, P. Bournai, M. Le Borgne, P. Le Guernic. A Design Environment for Discrete-Event Controllers based on the SIGNAL Language. In *1998 IEEE International Conf. On Systems, Man, And Cybernetics*, pp. 770-775, San Diego, California, USA, October 1998.

[C31] H. Marchand, M. Le Borgne. On the Optimal Control of Polynomial Dynamical Systems over Z/pZ. In *4th IEE International Workshop on Discrete Event Systems*, pp. 385-390, Cagliari, Italie, August 1998.

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[AČZ06] R. Alur, P. Černý, and S. Zdancewic. Preserving secrecy under refinement. In *ICALP '06: Proceedings (Part II) of the 33rd International Colloquium on Automata, Languages and Programming*, pages 107–118. Springer, 2006.

[AFF02] K. Akesson, H. Flordal, and M. Fabian. Exploiting modularity for synthesis and verification of supervisors. In *Proc. of the IFAC*, Barcelona, Spain, July 2002.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[AW02] S. Abdelwahed and W. Wonham. Supervisory control of interacting discrete event systems. In *41th IEEE Conference on Decision and Control*, pages 1175–1180, Las Vegas, USA, December 2002.

[BAF05] B. Blanchet, M. Abadi, and C. Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005. IEEE Computer Society.

[BBB+07] E. Badouel, M. Bednarczyk, A. Borzyszkowski, B. Caillaud, and P. Darondeau. Concurrent secrets. *Discrete Event Dynamic Systems*, 17(4):425–446, December 2007.

[BBG+10] S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *ATVA'10*, volume 6252 of *LNCS*, pages 52–66. Springer, 2010.

[BG92] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of qdds. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 172–186, London, UK, 1997. Springer-Verlag.

[BKMR08] J. Bryans, M. Koutny, L. Mazaré, and P. Ryan. Opacity generalised to transition systems. *International Journal of Information Security*, 7(6):421–435, 2008.

[BL00] G. Barrett and S. Lafortune. Decentralized supervisory control with communicating controllers. *IEEE Transactions on Automatic Control*, 45(9):1620–1638, 2000.

[BLL+05] N. Ben Hadj-Alouane, S. Lafrance, F. Lin, J. Mullins, and M. Yeddes. On the verification of intransitive noninterference in mulitlevel security. *IEEE Transaction On Systems, Man, And Cybernetics-Part B: Cybernetics*, 35(5):948–957, October 2005.

[BLPZ98] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella. Diagnosis of active systems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 274–278, 1998.

[BM95] S. K. Das F. Sarkar K. Basu and S. Madhavapeddy. Parallel discrete event simulation in star networks with application to telecommunications. In *MASCOTS '95: Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 66–71, 1995.

[BMD00] B. Brandin, R. Malik, and P. Dietrich. Incremental system verification and synthesis of minimally restrictive behaviours. In *Proceedings of the American Control Conference*, pages 4056–4061, Chicago, Illinois, June 2000.

[BMS10] B. Bérad, J. Mullins, and M. Sassolas. Quantifying opacity. In *Seventh International Conference on the Quantitative Evaluation of Systems (QEST)*, 2010.

[Bri88] E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification (PSTV'88)*, pages 63–74, 1988.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM computing Surveys*, pages 293–318, September 1992.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[Cao89] X-R. Cao. The predictability of discrete event systems. *IEEE Transactions on Automatic Control*, 34(11):1168–1171, 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252, 1977.

[CDFV88] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete-event process with partial observation. *IEEE Trans. on Automatic Control*, 33(3):249–260, 1988.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 470–475, Grenoble, France, avril 2002.

[CL08] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 2008.

[CLT04] O. Contant, S. Lafortune, and D. Teneketzis. Diagnosis of intermittent faults. *Discrete Event Dynamic Systems: Theory and Applications*, 14(2):171–202, 2004.

[CLT06] Olivier Contant, Stéphane Lafortune, and Demosthenis Teneketzis. Diagnosability of discrete event systems with modular structure. *Discrete Event Dynamic Systems*, 16(1):9–37, 2006.

[CR90] C. Chase and P. J. Ramadge. Predictability of a class of one-dimensional supervised systems. In *Proceedings of the Fifth IEEE International Symposium on Intelligent Control*, 1990.

[CT08] F. Cassez and S. Tripakis. Fault diagnosis with static and dynamic diagnosers. *Fundamenta Informaticae*, 88(4):497–540, November 2008.

[Dar05] P. Darondeau. Distributed implementations of Ramadge-Wonham supervisory control with petri nets. In *44th IEEE Conference on Decision and Control*, pages 2107–2112, Sevilla, Spain, December 2005.

[DC00a] M. H. De Queiroz and J. Cury. Modular control of composed systems. In *Proceedings of the American Control Conference*, pages 4051–4055, Chicago, Illinois, June 2000.

[dC00b] M.H. deQueiroz and J.E.R. Cury. Modular supervisory control of large scale discrete-event systems. In *Proc of 5th Workshop on Discrete Event Systems, WODES 2000*, pages 103–110, 2000.

[Dec98] P. Declerck. Predictability and control synthesis in time deviant graphs. In *Workshop on Discrete- Event Systems*, 1998.

[DFG+06] V. Darmaillacq, J-C. Fernandez, R. Groz, L. Mounier, and J-L. Richier. Test generation for network security rules. In *TestCom 2006*, volume 3964 of *LNCS*, 2006.

[DLT00a] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete-event systems. *Discrete Event Dynamic System : Theory and Applications*, 10(1/2):33–86, Janvier 2000.

[DLT00b] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. *Discrete Event Dynamical Systems: Theory and Applications*, 10:33–79, 2000.

[Dub09] J. Dubreil. *Monitoring and Supervisory Control for Opacity Properties*. PhD thesis, Université de Rennes 1, November 2009.

[FBJ+00] E. Fabre, A. Benveniste, C. Jard, L. Ricker, and M. Smith. Distributed state reconstruction for discrete event systems. In *IEEE Control and Decision Conference (CDC)*, Sydney, 2000.

[FFM12] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer (STTT)*, 14(3):349–382, 2012.

[FH99] H.K. Fadel and L.E. Holloway. Using spc and template monitoring method for fault detection and prediction in discrete event manufacturing systems. In *Proceedings of the 1999 IEEE International Symposium on Intelligent Control/Intelligent Systems and Semiotics*, pages 150–155, 1999.

[Fid88] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[FMFR11] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[Gau04] B. Gaudin. *Synthèse de contrôleurs sur des systèmes á évńements discrets structurs*. PhD thesis, Université de Rennes 1, Npvember 2004.

[Gen05] B. Genest. On implementation of global concurrent systems with local asynchronous controllers. In *CONCUR*, volume 3653 of *LNCS*, pages 443–457, 2005.

[GL06a] S. Genc and S. Lafortune. Diagnosis of patterns in partially-observed discrete-event systems. In *45th IEEE Conference on Decision and Control*, 2006.

[GL06b] S. Genc and S. Lafortune. Predictability in discrete-event systems under partial observation. In *IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*, 2006.

[GMM06] A. Genon, T. Massart, and C. Meuter. Monitoring distributed controllers: When an efficient ltl algorithm on sequences is needed to model-check traces. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2006.

[GSZ09] P. Gastin, N. Sznajder, and M. Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, 2009.

[Gun97] J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems*. PhD thesis, Linköping University, 1997.

[HMF06] K. W. Hamlen, G. Morrisett, and B. Schneider F. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[HR02] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 342–356, 2002.

[HWT92] G. Hoffmann and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *Proc. of 1992 American control Conference*, pages 2789–2793, Chicago,Il,USA, 1992.

[Ira09] K. Iraishi. On solvability of a decentralized supervisory control problem with communication. *IEEE Transactions on Automatic Control*, 54(3):468–480, March 2009.

[JHCK01] S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial time algorithm for diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 46(8):1318–1321, 2001.

[JK00] S. Jiang and R. Kumar. Decentralized control of discrete event systems with specializations to local control and concurrent systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 30(5):653–660, October 2000.

[JK04] S. Jiang and R. Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications. *IEEE Transactions on Automatic Control*, 49(6):934–945, 2004.

[JKG03] S. Jiang, R. Kumar, and H.E. Garcia. Diagnosis of repeated/intermittent failures in discrete event systems. *IEEE Transactions on Robotics and Automation*, 19(2):310–323, Avril 2003.

[JLF16] R. Jacob, J.-J. Lesage, and J.-M. Faure. Overview of Discrete Event Systems Opacity: models, validation, and quantification. *Annual Reviews in Control*, April 2016.

[JTGVR94] C. Jard, Jéron T, Jourdan G.-V, and J.-X. Rampon. A general approach to trace-checking in distributed computing systems. In *ICDCS*, pages 396–403, 1994.

[Kal10] G. Kalyon. *Supervisory control of infinite state systems under partial observation*. PhD thesis, Université Libre de Bruxelles, October 2010.

[Kar78] R Karp. A characterization of the minimum mean cycle in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[KGS91] R. Kumar, V. Garg, and Marcus S.I. On controllability and normality of discrete event dynamical systems. *Systems & Control Letters*, 17:157–169, 1991.

[KMMB07] G. Kalyon, T. Massart, C. Meuter, and L. Van Begin. Testing distributed systems through symbolic model checking. In *FORTE*, volume 4574 of *LNCS*, pages 263–279, 2007.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266. IEEE Computer Society, 1977.

[Kri63] S.A. Kripke. Semantical consideration on modal logic. *Acta Philisophica Fennica*, 16:83–94, 1963.

[KvS05] J. Komenda and J.H. van Schuppen. Supremal sublanguages of general specification languages arising in modular control of discrete-event systems. In *44th IEEE Conference on Decision and Control*, pages 2775–2780, 2005.

[KvS08] J. Komenda and J. "van Shuppen. Coordination control of discrete-event systems. In *Workshop on Discrete Event Systems, WODES'08*, Gothenburg, Sweden, March 2008.

[KWKL16] C. Kawakami, Y. Wu, R. Kwong, and S. Lafortune. Detection and prevention of actuator enablement attacks in supervisory control systems. In *Proc of 13th Workshop on Discrete Event Systems, WODES'16*, 2016.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LBW05] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, February 2005.

[LBW09] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41, 2009.

[Le 07] G. Le Guernic. Information flow testing - the third path towards confidentiality guarantee. In *Advances in Computer Science - ASIAN 2007. Computer and Network Security, LNCS No 4846*, pages 33–47, 2007.

[LGJJ06] T. Le Gall, B. Jeannet, and T. Jéron. Verification of communication protocols using abstract interpretation of fifo queues. In *11th International Conference on Algebraic Methodology and Software Technology, AMAST '06*, LNCS, July 2006.

[LK02] S.-H. Lee and Wong K.C. Structural decentralized control of concurrent discrete-event systems. *European Journal of Control*, 8(5), 2002.

[LLW05] R.J. Leduc, M Lawford, and W.M. Wonham. Hierarchical interface-based supervisory control-part ii: parallel case. *IEEE Transactions on Automatic Control*, 50(9):1336–1348, September 2005.

[Low99] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(2-3):89–146, 1999.

[LRL07] F. Lin, K. Rudie, and S. Lafortune. Minimal communication for essential transitions in a distributed discrete-event system. *IEEE Transactions on Automatic Control*, 52(8):1495–1502, June 2007.

[LS95] F. Laroussinie and P. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148(2):303–324, September 1995.

[Mar75] D. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975.

[Mar97] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Higher Education, 1997.

[Mas91] T. Massart. A calculus to define correct tranformations of lotos specifications. In *FORTE*, volume C-2 of *IFIP Transactions*, pages 281–296, 1991.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[Maz86] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324, 1986.

[Maz04] L. Mazaré. Using unification for opacity properties. In *Proceedings of the 4th IFIP WG1.7 Workshop on Issues in the Theory of Security (WITS'04)*, pages 165–176, Barcelona (Spain), 2004.

[McS10] McScM, a Model Checker for Symbolic Communicating Machines - version 1.2, 2010. http://altarica.labri.fr/forge/projects/mcscm/wiki/.

[Mil81] R. Milner. A modal characterisation of observable machine-behaviour. In *CAAP*, pages 25–34, 1981.

[MM06] M. Musolesi and C. Mascolo. Evaluating context information predictability for autonomic communication. In *American Control Conference*, 2006.

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT '72: Proceedings of the 13th Annual Symposium on Switching and Automata Theory (swat 1972)*, pages 125–129, Washington, DC, USA, 1972.

[MW06] C. Ma and M. Wonham. Nonblocking supervisory control of state tree structures. *IEEE Transactions on Automatic Control*, 51(5):782–793, 2006.

[MW07] A. Muscholl and I. Walukiewicz. A lower bound on web services composition. In *FOSSACS'07*, pages 274–286, Berlin, Heidelberg, 2007. Springer-Verlag.

[PC05] Y. Pencolé and M-O. Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence Journal*, 164(1-2):121–170, 2005.

[Pin15] B. Pinisetty. *Runtime validation of critical control-command systems*. PhD thesis, Université de Rennes 1, January 2015.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[PP91] W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM Trans. Program. Lang. Syst.*, 13(3):399–442, July 1991.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society, 1990.

[QHF02] G. Riley Q. He, M. Ammar and R. Fujimoto. Exploiting the predictability of tcp's steady-state behavior to speed up network simulation. In *10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 101–108, 2002.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RC98] L. Rozé and M.-O. Cordier. Diagnosing discrete-event systems : an experiment in telecommunication networks. In *4th International Workshop on Discrete Event Systems*, pages 130–137, 1998.

[RC11] L. Ricker and B. Caillaud. Mind the gap: Expanding communication options in decentralized discrete-event control. *Automatica*, 47(11):2364 – 2372, 2011.

[Ric00] K. Ricker, L. Rudie. Know means no: Incorporating knowledge into discrete-event control systems. *IEEE Transactions on Automatic Control*, 45(9):1656–1668, September 2000.

[RL03] K. Rohloff and S. Lafortune. The control and verification of similar agents operating in a broadcast network environment. In *42nd IEEE Conference on Decision and Control*, 2003.

[RV01] L. Ricker and J. Van Shuppen. Decentralized failure diagnosis with asynchronous communication between supervisors. In *Proceedings of 2001 IEEE European Control Conference*, September 2001.

[RW89] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[RW92] K. Rudie and W.M. Wonham. Think globally, act locally: decentralized supervisory control. *IEEE Transaction on Automatic Control*, 31(11):1692–1708, November 1992.

[Sch00a] F. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[Sch00b] F. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SG02] A. Sen and V. K. Garg. Detecting temporal logic predicates on the happened-before model. In *IPDPS*, 2002.

[SH07] A. Saboori and C. N. Hadjicostis. Notions of security and opacity in discrete event systems. In *CDC'07:* 46^{th} *IEEE Conf. Decision and Control*, pages 5056–5061, 2007.

[SH13] A. Saboori and C. Hadjicostis. Verification of initial-state opacity in security applications of discrete event systems. *Inf. Sci.*, 246:115–132, 2013.

[SRBT91] C. Papadimitriou S. R. Buss and J. Tsitsiklis. On the predictability of coupled automata: an allegory about chaos. *Complex Systems*, 5:525–539, 1991.

[SSL+95] M. Sampath, R. Sengupta, S. Lafortune, K. Sinaamohideen, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

[SSL+96] M. Sampath, R. Sengupta, S. Lafortune, K. Sinaamohideen, and D. Teneketzis. Failure diagnosis using discrete event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, Mars 1996.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TML+16] Y. Tong, Z. Ma, Z. Li, C. Seatzu, and A. Giua. Supervisory enforcement of current-state opacity with uncomparable observations. In *Proc of 13th Workshop on Discrete Event Systems, WODES'16*, 2016.

[TO08] S. Takai and Y. Oka. A formula for the supremal controllable and opaque sublanguage arising in supervisory control. *SICE Journal of Control, Measurement, and System Integration*, 1(4):307–312, March 2008.

[Tre96a] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[Tre96b] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tri01] S. Tripakis. Undecidable problems of decentralized control and observation. In *Proceedings of 2001 IEEE Conference on Decision and Control*, 2001.

[Tri04] S. Tripakis. Decentralized control of discrete event systems with bounded or unbounded delay communication. *IEEE Trans. on Automatic Control*, 49(9):1489–1501, 2004.

[VLF04] A. Vahidi, B. Lennarston, and M. Fabian. Efficient supervisory synthesis of large systems. In *Proc of 7th Workshop on Discrete Event Systems, WODES'04*, 2004.

[WH91] Y. Willner and M. Heymann. Supervisory control of concurrent discrete-event systems. *International Journal of Control*, 54(5):1143–1169, 1991.

[WL12] Y. Wu and S. Lafortune. Enforcement of opacity properties using insertion functions. In *51st IEEE Conf. on Decision and Contr.*, pages 6722–6728, 2012.

[Won03] W. M. Wonham. Notes on control of discrete-event systems. Technical Report ECE 1636F/1637S, Department of ECE, University of Toronto, July 2003.

[WR88] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of Control Signals and Systems*, 1:13–30, 1988.

[XK09] S. Xu and R. Kumar. Distributed state estimation in discrete event systems. In *ACC'09: Proceedings of the 2009 conference on American Control Conference*, pages 4735–4740. IEEE Press, 2009.

[YL00] T. Yoo and S. Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. In *Proc of 5th Workshop on Discrete Event Systems, WODES 2000*, Ghent, Belgium, August 2000.

[YL02] T. Yoo and S. Lafortune. Polynomial-time verification of diagnosability of partially-observed discrete-event systems. *IEEE Trans. on Automatic Control*, 47(9):1491–1495, Septembre 2002.

[YP10] L. Ye and Dague; P:. An optimized algorithm for diagnosability of component-based systems. In *10th IFAC Workshop on Discrete Event Systems*, pages 143–148, Berlin, Germany, avril 2010.

[ZL13] J. Zaytoon and S. Lafortune. Overview of fault diagnosis methods for discrete event systems. *Annual Reviews in Control*, 37(2):308–320, 2013.

[ZP96] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.

Science of Computer Programming 123 (2016) 2–41

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Runtime enforcement of regular timed properties by suppressing and delaying events

Yliès Falcone [a,*], Thierry Jéron [b], Hervé Marchand [b], Srinivas Pinisetty [c]

[a] Univ. Grenoble Alpes, INRIA, LIG, F-38000 Grenoble, France
[b] INRIA Rennes – Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France
[c] Aalto University, Finland

ARTICLE INFO

Article history:
Received 2 September 2014
Received in revised form 26 January 2016
Accepted 26 February 2016
Available online 4 March 2016

Keywords:
Verification
Monitoring
Runtime enforcement
Timed specifications

ABSTRACT

Runtime enforcement is a verification/validation technique aiming at correcting possibly incorrect executions of a system of interest. In this paper, we consider enforcement monitoring for systems where the physical time elapsing between actions matters. Executions are thus modelled as timed words (i.e., sequences of actions with dates). We consider runtime enforcement for timed specifications modelled as timed automata. Our enforcement mechanisms have the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus possibly allowing for longer executions. To ease their design and their correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that specify the input–output behaviour in terms of transformations of timed words, constraints that should be satisfied by such functions, enforcement monitors that describe the operational behaviour of enforcement functions, and enforcement algorithms that describe the implementation of enforcement monitors. The feasibility of enforcement monitoring for timed properties is validated by prototyping the synthesis of enforcement monitors from timed automata.

1. Introduction

Runtime enforcement [1–5] is a verification and validation technique aiming at correcting possibly-incorrect executions of a system of interest. In traditional (untimed) approaches, the enforcement mechanism is a *monitor* modelled as a transducer that inputs, corrects, and outputs a sequence of events. How a monitor transforms the input sequence is done according to a specification of correct sequences, formalised as a property. Moreover, a monitor should satisfy some requirements: it should be *sound* in the sense that only (prefixes of) correct sequences are output; it should also be *transparent* meaning that the output sequence preserves some relation with the input sequence, depending on the authorised operations.

Runtime enforcement monitors can be used in various application domains. For instance, enforcement monitors can be used for the design of firewalls, to verify the control-flow integrity and memory access of low-level code [6], or implemented in security kernels or virtual machines to protect the access to sensitive system resources (e.g., [7]). In [8], we discuss some other uses of enforcement monitors such as resource allocation and the implementation of robust mail servers.

* Corresponding author.
E-mail addresses: ylies.falcone@imag.fr (Y. Falcone), thierry.jeron@inria.fr (T. Jéron), herve.marchand@inria.fr (H. Marchand), srinivas.pinisetty@aalto.fi (S. Pinisetty).

http://dx.doi.org/10.1016/j.scico.2016.02.008
0167-6423/

In this paper, we consider *runtime enforcement of timed properties*, initially introduced in [5,9]. In timed properties (over finite sequences), not only the order of events matters, but also their occurrence dates affect the satisfaction of the property. It turns out that considering time constraints when specifying the behaviour of systems brings some expressiveness that can be particularly useful in some application domains when, for instance, specifying the usage of resources. In Section 2, we present some running and motivating examples of timed specifications related to the access of resources by processes. We shall see that, in contrast to the untimed case, the amount of time an event is stored influences the satisfaction of properties.

In [5], we propose preliminary enforcement mechanisms restricted to safety and co-safety timed properties. Safety and co-safety properties allow to express that "something bad should never happen" and that "something good should happen within a finite amount of time", respectively. In [9], we generalise and extend the initial approach of [5] to the whole class of timed regular properties. Indeed, some regular properties may express interesting behaviours of systems belonging to a larger class that allows to specify some form of transactional behaviour. Regular properties are, in general, neither prefix nor extension closed, meaning that the evaluation of an input sequence w.r.t. the property also depends on its possible future continuations. For instance, an incorrect input sequence alone may not be correctable by an enforcement mechanism, but the reception of some events in the future may allow some correction. Hence, the difficulty that arises is that the enforcement mechanism should take conservative decisions and change its behaviour over time taking into account the evaluation (w.r.t. the property) of the current input sequence and its possible continuations. Roughly speaking, in [5,9], enforcement mechanisms receive sequences of events composed of actions and delays between them, and can only increase those delays to satisfy the desired timed property; while in this paper, we consider absolute dates and allow to reduce delays between events (as described in detail in the following paragraph).

Contributions In this paper, we extend [9] in several directions. The main extension consists in increasing the power of enforcement mechanisms by allowing them to suppress input events, when the monitor determines that it is not possible to correct the input sequence, whatever is its continuation. Consequently, enforcement mechanisms can continue operating, and outputting events, while in our previous approaches the output would have been blocked forever. This feature and other considerations also drove us to revisit and simplify the formalisation of enforcement mechanisms. We now consider events composed of actions and absolute dates, and enforcement mechanisms are *time retardant with suppression* in the following sense: monitors should keep the same order of the actions that are not suppressed, and are allowed to increase the absolute dates of actions in order to satisfy timing constraints. Note, this allows to decrease delays between actions, while it is not allowed in [5,9]. As in [5,9], we specify the mechanisms at several levels, but in a revised and simplified manner: the notion of enforcement function describes the behaviour of an enforcement mechanism at an abstract level as an input–output relation between timed words; requested properties of these functions are formalised as soundness, transparency, optimality, and additional physical constraints[1]; we design adequate enforcement functions and prove that they satisfy those properties; the operational behaviour of enforcement functions is described as enforcement monitors, and it is proved that those monitors correctly implement the enforcement functions; finally enforcement algorithms describe the implementation of enforcement monitors and serve to guide the concrete implementation of enforcement mechanisms. Interestingly, although all untimed regular properties over finite sequences can be enforced [10], some enforcement limitations arise for timed properties (over finite sequences). Indeed, we show that storing events in the timed setting influences the output of enforcement mechanisms. In particular, because of physical time, an enforcement mechanism might not be able to output certain correct input sequences. Finally, we propose an implementation of the enforcement mechanisms for all regular properties specified by one-clock timed automata (while [5,9] feature an implementation for safety and co-safety properties only).

Paper organisation The rest of this paper is organised as follows. In Section 2, we introduce some motivating and running examples for the enforcement monitoring of timed properties, and illustrate the behaviour of enforcement mechanisms and the enforceability issues that arise. Section 3 introduces some preliminaries and notations. Section 4 recalls timed automata. Section 5 introduces our enforcement monitoring framework and specifies the constraints that should be satisfied by enforcement mechanisms. Section 6 defines enforcement functions as functional descriptions of enforcement mechanisms. Section 7 defines enforcement monitors as operational description of enforcement mechanisms in the form of transition systems. Section 8 proposes algorithms that effectively implement enforcement monitors. In Section 9, we present an implementation of enforcement mechanism in Python and evaluate the performance of synthesised enforcement mechanisms. In Section 10, we discuss related work. In Section 11, we draw conclusions and open perspectives. Finally, to ease the reading of this article, some proofs are sketched and their complete versions can be found in Appendix A.

2. General principles and motivating examples

In this section, we describe the general principles of enforcement monitoring of timed properties, and illustrate the expected input/output behaviour of enforcement mechanisms on several examples.

[1] The two latter constraints are specific to runtime enforcement of timed properties.

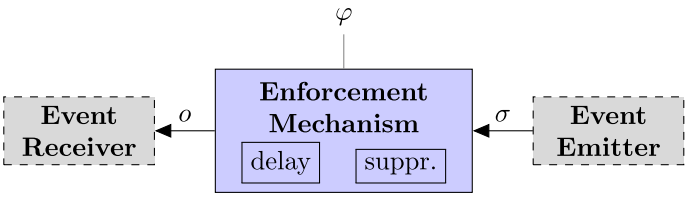


Fig. 1. Illustration of the principle of enforcement monitoring.

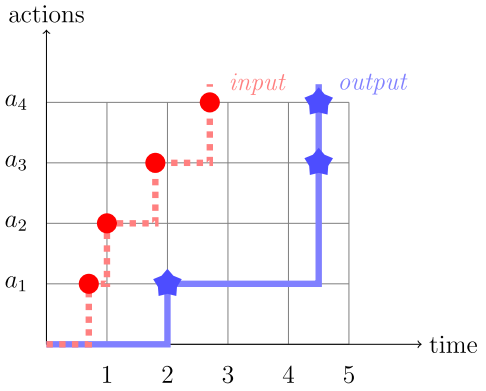


Fig. 2. Behaviour of an enforcement mechanism.

2.1. General principles of enforcement monitoring in a timed context

As illustrated in Fig. 1, the purpose of enforcement monitoring is to read some (possibly incorrect) input sequence of events σ produced by a system, referred to as the event emitter, to transform it into an output sequence of events o that is correct w.r.t. a specification formalised by a property φ. This output sequence is then transmitted to an event receiver. In our timed setting, events are actions with their occurrence dates. Input and output sequences of events are then formalised by timed words and enforcement mechanisms can be seen as transformers of timed words.

Fig. 2 illustrates the behaviour of an enforcement mechanism when correcting an input sequence. The dashed and solid curves respectively represent input and output sequences of events (occurrence dates in abscissa and actions in ordinate). The behaviour of an enforcement mechanism should satisfy some constraints, namely *physical constraint*, *soundness*, and *transparency*. Intuitively, the physical constraint states that an enforcement mechanism cannot modify what it has already output, i.e., the output forms a continuously-growing sequence of events; soundness states that the output sequence should be correct w.r.t. the property (note, soundness is not represented in the figure, since this would require to represent an area containing only the sequences admitted by the property); transparency states that the output sequence is obtained by delaying or suppressing actions from the input sequence (and not changing the order of actions); thus, if the events of the input curve are not suppressed, they appear later in the output curve, in the same order. For example, actions a_1, a_3 and a_4 are delayed but a_2 is suppressed. Notice that by delaying dates of events the enforcement mechanism allows to reduce delays between events. For example, action a_4 occurs strictly after action a_3, but both actions are released at the same date. Moreover, the actions should be released as output as soon as possible, which will be described by an *optimality property*.

2.2. Motivating examples

We introduce some running and motivating examples related to the usage of resources by some processes. We also provide some intuition on the expected behaviour of our enforcement mechanisms, and point out some issues arising in the timed context. We discuss further these issues and their relation to the expected constraints on enforcement mechanisms.

Let us consider the situation where two processes access to and operate on a common resource. Each process i (with $i \in \{1, 2\}$) has three interactions with the resource: acquisition (acq_i), release (rel_i), and a specific operation (op_i). Both processes can also execute a common action op. System initialisation is denoted by action $init$. In the following, variable t keeps track of global time. Figs. 3, 4, and 5 illustrate the behaviour of enforcement mechanisms for several specifications on the behaviour of the processes and for particular input sequences.[2]

[2] We shall see in Section 3.2 how to formalise these specifications by timed automata.

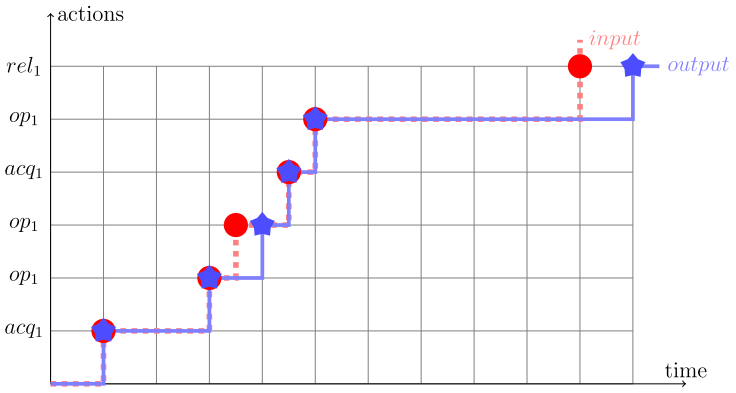


Fig. 3. Behaviour of an enforcement mechanism for specification S_1 on σ_1.

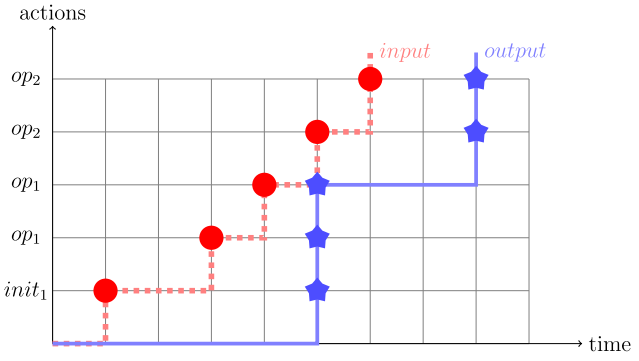


Fig. 4. Behaviour of an enforcement mechanism for specification S_2 on σ_2.

Specification S_1 The specification states that "*Each process should acquire the resource before performing operations on it and should release it afterwards. Each process should keep the resource for at least 10 time units (t.u.). There should be at least 1 t.u. between any two operations.*"

Let us consider the input sequence $\sigma_1 = (1, acq_1) \cdot (3, op_1) \cdot (3.5, op_1) \cdot (4.5, acq_1) \cdot (5, op_1) \cdot (10, rel_1)$ (where each event is composed of an action associated with a date, indicating the time instant at which the action is received as input). The monitor receives the first action acq_1 at $t = 1$, followed by op_1 at $t = 3$, etc. At $t = 1$ (resp. $t = 3$), the monitor can output action acq_1 (resp. op_1) because both sequences $(3, op_1)$ and $(1, acq_1) \cdot (3, op_1)$ satisfy specification S_1. At $t = 3.5$, when the second action op_1 is input, the enforcer determines that this action should be delayed by 0.5 t.u. to ensure the constraint that 1 t.u. should elapse between occurrences of op_1 actions. Hence, the second action op_1 is released at $t = 4$. At $t = 4.5$, when action acq_1 is received, the enforcer releases it immediately since this action is allowed by the specification with no time constraint. Similarly, at $t = 5$, an op_1 action is received and is released immediately because at least 1 t.u. elapsed since the previous op_1 action was released as output. At $t = 10$, when action rel_1 is received, it is delayed by 1 t.u. to ensure that the resource is kept for at least 10 t.u. (the first acq_1 action was released at $t = 1$). Henceforth, as shown in Fig. 3, the output of the enforcement mechanism for σ_1 is $(1, acq_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (4.5, acq_1) \cdot (5, op_1) \cdot (11, rel_1)$.

Specification S_2 The specification states that "*After system initialisation, both processes should perform an operation (actions op_i) before 10 t.u. The operations of the different processes should be separated by 3 t.u.*"

Let us consider the input sequence $\sigma_2 = (1, init_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (5, op_2) \cdot (6, op_2)$. At $t = 1, 3, 4$, when the enforcement mechanism receives the actions, it cannot release them as output but memorises them since, upon each reception, the sequence of actions it received so far cannot be delayed so that a known continuation may satisfy specification S_2. At $t = 5$, upon the reception of action op_2, the sequence received so far can be delayed to satisfy specification S_2. Action $init_1$ is released at $t = 5$ because it is the earliest possible date: a smaller date would be already elapsed. The two actions op_1 are also released at $t = 5$, because there are no timing constraints on them. The first action op_2 is released at $t = 8$ to ensure a delay of at least 3 t.u. with the first op_1 action. The second action op_2, received at $t = 6$, is also released at $t = 8$, since it does not need to be delayed more than after the preceding action. Henceforth, as shown in Fig. 4, the output of the enforcement mechanism for σ_2 is $(5, init_1) \cdot (5, op_1) \cdot (5, op_1) \cdot (8, op_2) \cdot (8, op_2)$.

Specification S_3 The specification states that "*Operations op_1 and op_2 should execute in a transactional manner. Both actions should be executed, in any order, and any transaction should contain one occurrence of op_1 and op_2. Each transaction should com-*

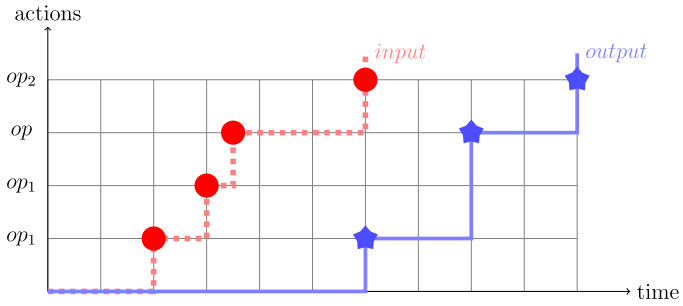


Fig. 5. Behaviour of an enforcement mechanism for specification S_3 on σ_3.

plete within 10 t.u. Between operations op_1 and op_2, occurrences of operation op can occur. There is at least 2 t.u. between any two occurrences of any operation."

Let us consider the input sequence $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. At $t = 2$, the monitor cannot output action op_1 because this action alone does not satisfy the specification (and the monitor does not yet know the next events i.e., actions and dates). If the next action was op_2, then, at the date of its reception, the monitor could output action op_1 followed by op_2, as it could choose dates for both actions in order to satisfy the timing constraints. At $t = 3$ the monitor receives a second op_1 action. Clearly, there is no possible date for these two op_1 actions to satisfy specification S_3, and no continuation could solve the situation. The monitor thus suppresses the second op_1 action, since this action is the one that prevents satisfiability in the future. At $t = 3.5$, when the monitor receives action op, the input sequence still does not satisfy the specification, but there exists an appropriate delaying of such action so that with future events, the specification can be satisfied. At $t = 6$, the monitor receives action op_2, it can decide that action op_1 followed by op and op_2 can be released as output with appropriate delaying. Thus, the date associated with the first op_1 action is set to 6 (the earliest possible date, since this decision is taken at $t = 6$), 8 for action op (since 2 is the minimal delay between those actions satisfying the timing constraint), and 10 for action op_2. Henceforth, as shown in Fig. 5, the output of the enforcer for σ_3 is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$.

Specification S_4 The specification states that "*Processes should behave in a transactional manner, where each transaction consists of an acquisition of the resource, at least one operation on it, and then its release. After the acquisition of the resource, the operations on the resource should be done within 10 t.u. The resource should not be released less than 10 t.u. after acquisition. There should be no more than 10 t.u. without any ongoing transaction.*"

Let us consider the input sequence $\sigma_4 = (1, acq_i) \cdot (2, op_i) \cdot (3, rel_i)$. Before $t = 3$, no output can be produced, since no transaction is complete, and events must be stored. At $t = 3$, when the monitor receives rel_i, it can decide that the three events acq_i, op_i, and rel_i can be released as output with appropriate delaying. Thus, the date associated with the two first actions acq_i and op_i is set to 3, since this is the minimal decision date. Moreover, to satisfy the timing constraint on release actions after acquisitions, the date associated to the last event rel_i is set to 13. The output of the enforcement mechanism for σ_4 is then $(3, acq_i) \cdot (3, op_i) \cdot (13, rel_i)$.

Let us now consider the input sequence $\sigma'_4 = (3, acq_i) \cdot (7, op_i) \cdot (13, rel_i)$. The monitor observes action acq_i followed by an op_i and a rel_i actions only at date $t = 13$. Hence, the date associated with the first action in the output should be at least 13, which is the minimal decision date. However, if the monitor chooses a date for acq_i which is strictly greater than 10, the timing constraint cannot be satisfied. Consequently, the output of the monitor remains always empty. Notice however that the input sequence provided to the monitor satisfies the specification. Nevertheless, the monitor cannot release any event as output as it cannot take a decision until it receives action rel_i at date $t = 13$, which affects the date (i.e., the absolute time instant when it can be released as output) of the first action acq_i, thus falsifying the constraints.

Discussion Specification S_4 illustrates an important issue of enforcement in the timed setting, exhibited in this paper: because input timed words are seen as streams of events with dates, for some properties, there exist some input timed words that cannot be enforced, even though they either already satisfy the specification, or could be delayed to satisfy the specification (if they were known in advance). For instance, we shall see that specifications S_1, S_2, and S_3 do not suffer from this issue, while S_4 does. Actually, it turns out that enforcement monitors face some constraints due to streaming: they need to memorise input timed events before taking decision, but meanwhile, time elapses and this influences the possibility to satisfy the considered specification. Nevertheless, the synthesis of enforcement mechanisms proposed in this paper works for all regular timed properties, which means that the synthesised enforcement mechanisms still satisfy their requirements (soundness, transparency, optimality, and physical constraint), even though the output may be empty for some input timed words.

3. Preliminaries and notation

We first recall some basic notions on untimed languages (Section 3.1). We then introduce timed words and languages (Section 3.2) and extend previous notions in a timed setting (Section 3.2). Finally, we introduce some orders on timed words that will be used in runtime enforcement (Section 3.3).

3.1. Untimed languages

A (finite) word over an alphabet A is a finite sequence $w = a_1 \cdot a_2 \cdots a_n$ of elements of A. The *length* of w is n and is noted $|w|$. The empty word over A is denoted by ϵ_A, or ϵ when clear from the context. The set of all (respectively non-empty) words over A is denoted by A^* (respectively A^+). A *language* over A is any subset $\mathcal{L}$ of A^*.

The *concatenation* of two words w and w' is noted $w \cdot w'$. A word w' is a *prefix* of a word w, noted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$, and $w' \prec w$ if additionally $w' \neq w$; conversely w is said to be an *extension* of w'.

The set $\text{pref}(w)$ denotes the *set of prefixes* of w and subsequently, $\text{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \bigcup_{w \in \mathcal{L}} \text{pref}(w)$ is the set of prefixes of words in $\mathcal{L}$. A language $\mathcal{L}$ is *prefix-closed* if $\text{pref}(\mathcal{L}) = \mathcal{L}$ and *extension-closed* if $\mathcal{L} \cdot A^* = \mathcal{L}$.

Given two words u and v, $v^{-1} \cdot u$ is the *residual* of u by v and denotes the word w, such that $v \cdot w = u$, if this word exists, i.e., if v is a prefix of u. Intuitively, $v^{-1} \cdot u$ is the suffix of u after reading prefix v. By extension, for a language $\mathcal{L} \subseteq A^*$ and a word $v \in A^*$, the residual of $\mathcal{L}$ by v is the language $v^{-1} \cdot \mathcal{L} \stackrel{\text{def}}{=} \{w \in A^* \mid v \cdot w \in \mathcal{L}\}$. It is the set of suffixes of words that, concatenated to v, belong to $\mathcal{L}$. In other words, $v^{-1} \cdot \mathcal{L}$ is the set of suffixes of words in $\mathcal{L}$ after reading prefix v.

For a word w and $i \in [1, |w|]$, the i-th letter of w is noted $w_{[i]}$. Given a word w and two integers i, j, s.t. $1 \leq i \leq j \leq |w|$, the *subword* from index i to j is noted $w_{[i \cdots j]}$.

Given two words w and w', we say that w' *is a subsequence of* w, noted $w' \lhd w$, if there exists an increasing mapping $k : [1, |w'|] \to [1, |w|]$ (i.e., $\forall i, j \in [1, |w'|] : i < j \implies k(i) < k(j)$) such that $\forall i \in [1, |w'|] : w'_{[i]} = w_{[k(i)]}$. Notice that, k being increasing entails that $|w'| \leq |w|$. Intuitively, the image of $[1, |w'|]$ by function k is the set of indexes of letters of w that are "kept" in w'.

Given an n-tuple of symbols $e = (e_1, \ldots, e_n)$, for $i \in [1, n]$, $\Pi_i(e)$ is the projection of e on its i-th element ($\Pi_i(e) \stackrel{\text{def}}{=} e_i$). Operator Π_i is naturally extended to sequences of n-tuples of symbols to produce the sequence formed by the concatenation of the projections on the i-th element of each tuple.

3.2. Timed words and languages

As sketched in Section 2, input and output streams are seen as sequences of events composed of a date and an action, where the date is interpreted as the absolute date when the action is received by the enforcement mechanism. In what follows, we formalise input and output streams with timed words, and related notions, generalising the untimed setting.

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers, and Σ a finite alphabet of *actions*. An *event* is a pair $(t, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, where $\text{date}((t, a)) \stackrel{\text{def}}{=} t \in \mathbb{R}_{\geq 0}$ is the absolute time instant at which action $\text{act}((t, a)) \stackrel{\text{def}}{=} a \in \Sigma$ occurs.

A timed word over alphabet Σ is a finite sequence of events $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$, where $(t_i)_{i \in [1,n]}$ is a non-decreasing sequence in $\mathbb{R}_{\geq 0}$. We denote by $\text{start}(\sigma) \stackrel{\text{def}}{=} t_1$ the starting date of σ and $\text{end}(\sigma) \stackrel{\text{def}}{=} t_n$ its ending date (with the convention that the starting and ending dates are equal to 0 for the empty timed word ϵ).

The set of timed words over Σ is denoted by $\text{tw}(\Sigma)$. A *timed language* is any set $\mathcal{L} \subseteq \text{tw}(\Sigma)$. Note that even though the alphabet $(\mathbb{R}_{\geq 0} \times \Sigma)$ is infinite in this case, previous notions and notations defined in the untimed case (related to length, prefix, subword, subsequence etc) naturally extend to timed words.

The concatenation of timed words however requires more attention, as when concatenating two timed words, one should ensure that the result is a timed word, i.e., dates should be non-decreasing. This is ensured as soon as the ending date of the first timed word does not exceed the starting date of the second one. Formally, let $\sigma = (t_1, a_1) \cdots (t_n, a_n)$ and $\sigma' = (t'_1, a'_1) \cdots (t'_m, a'_m)$ be two timed words with $\text{end}(\sigma) \leq \text{start}(\sigma')$, their concatenation is $\sigma \cdot \sigma' \stackrel{\text{def}}{=} (t_1, a_1) \cdots (t_n, a_n) \cdot (t'_1, a'_1) \cdots (t'_m, a'_m)$. By convention $\sigma \cdot \epsilon \stackrel{\text{def}}{=} \epsilon \cdot \sigma \stackrel{\text{def}}{=} \sigma$. Concatenation is undefined otherwise.

The *untimed projection* of σ is $\Pi_\Sigma(\sigma) \stackrel{\text{def}}{=} a_1 \cdot a_2 \cdots a_n$ in Σ^* (i.e., dates are ignored).

Given $t \in \mathbb{R}_{\geq 0}$, and a timed word $\sigma \in \text{tw}(\Sigma)$, we define the *observation of* σ *at date* t as the prefix of σ that can be observed at date t. It is defined as the maximal prefix of σ whose ending date is lower than t:

$$\text{obs}(\sigma, t) \stackrel{\text{def}}{=} \max_{\preccurlyeq} \left\{ \sigma' \in \text{pref}(\sigma) \mid \text{end}(\sigma') \leq t \right\}.$$

3.3. Preliminaries to runtime enforcement

Apart from the prefix order $\preccurlyeq$ (defined in Section 3.1), the following partial orders on timed words will be useful for enforcement.

Delaying order $\succcurlyeq_d$ For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, we say that σ' *delays* σ (noted $\sigma' \succcurlyeq_d \sigma$) iff they have the same untimed projection but the dates of events in σ' exceed the dates of corresponding events in σ. Formally:

$$\sigma' \succcurlyeq_d \sigma \stackrel{\text{def}}{=} \Pi_\Sigma(\sigma') = \Pi_\Sigma(\sigma) \wedge \forall i \in [1, |\sigma|] : \mathrm{date}(\sigma'_{[i]}) \geq \mathrm{date}(\sigma_{[i]}).$$

Sequence σ' is obtained from σ by keeping all actions, but with a potential increase in dates.

For example, $(4,a)\cdot(7,b)\cdot(9,c) \succcurlyeq_d (3,a)\cdot(5,b)\cdot(8,c)$. Note that delays between events may be decreased, e.g., between b and c, but absolute dates are increased.

Delaying subsequence order $\lhd_d$ For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, we say that σ' *is a delayed subsequence of* σ (noted $\sigma' \lhd_d \sigma$) iff there exists a subsequence σ'' of σ such that σ' delays σ''. Formally:

$$\sigma' \lhd_d \sigma \stackrel{\text{def}}{=} \exists \sigma'' \in \mathrm{tw}(\Sigma) : \left(\sigma'' \lhd \sigma \wedge \sigma' \succcurlyeq_d \sigma''\right).$$

Sequence σ' is obtained from σ by first suppressing some actions, and then increasing the dates of the actions that are kept. This order will be used to characterise output timed words with respect to input timed words in enforcement monitoring when suppressing and delaying events.

For example, $(4,a)\cdot(9,c) \lhd_d (3,a)\cdot(5,b)\cdot(8,c)$ (event $(5,b)$ has been suppressed while a and c are shifted in time).

Lexical order $\preceq_{\text{lex}}$ This order is useful to choose a unique timed word among some with same untimed projection. For two timed words σ, σ' with same untimed projection (i.e., $\Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$), the order $\preceq_{\text{lex}}$ is defined inductively as follows: $\epsilon \preceq_{\text{lex}} \epsilon$, and for two events with identical actions (t,a) and (t',a), $(t,a)\cdot\sigma \preceq_{\text{lex}} (t',a)\cdot\sigma'$ if $t \leq t' \vee (t = t' \wedge \sigma \preceq_{\text{lex}} \sigma')$. For example $(3,a)\cdot(5,b)\cdot(8,c)\cdot(11,d) \preceq_{\text{lex}} (3,a)\cdot(5,b)\cdot(9,c)\cdot(10,d)$.

Choosing a unique timed word with minimal duration $\min_{\preceq_{\text{lex}},\text{end}}$ Given a set of timed words with same untimed projection, $\min_{\preceq_{\text{lex}},\text{end}}$ selects the minimal timed word w.r.t. the lexical order among timed words with minimal ending date: first the set of timed words with minimal ending date are considered, and then, from these timed words, the (unique) minimal one is selected w.r.t. the lexical order. Formally, for a set $E \subseteq \mathrm{tw}(\Sigma)$ such that $\forall \sigma, \sigma' \in E : \Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$ (i.e., such that all words have the same untimed projection), we have $\min_{\preceq_{\text{lex}},\text{end}}(E) = \min_{\preceq_{\text{lex}}}\left(\min_{\preceq_{\text{end}}}(E)\right)$ where $\sigma \preceq_{\text{end}} \sigma'$ if $\mathrm{end}(\sigma) \leq \mathrm{end}(\sigma')$, for $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$.

4. Timed languages and properties as timed automata

Timed automata is a usual model used to specify properties of sequences of events where timing between them matters. In this section, we introduce timed automata as a specification formalism for timed properties (Section 4.1). We describe a partitioning of the states of timed automata (Section 4.2). The partitioning allows to distinguish behaviours according to i) whether they currently satisfy or violate the property, and ii) whether or not this remains true for future behaviours. Finally, we present some sub-classes of regular properties (Section 4.3).

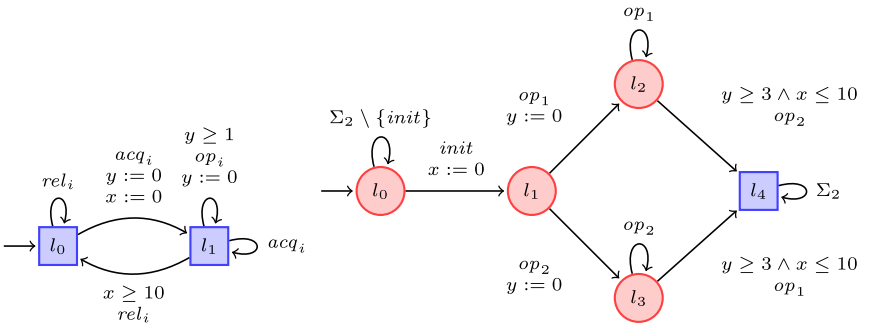
4.1. Timed automata

A timed automaton [11] is a finite automaton extended with a finite set of real valued clocks. Let $X = \{x_1, \ldots, x_k\}$ be a finite set of *clocks*. A *clock valuation* for X is an element of $\mathbb{R}^X_{\geq 0}$, that is, a function from X to $\mathbb{R}_{\geq 0}$. For $\nu \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ is the valuation assigning $\nu(x) + \delta$ to each clock x of X. Given a set of clocks $X' \subseteq X$, $\nu[X' \leftarrow 0]$ is the clock valuation ν where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of *guards*, i.e., clock constraints defined as Boolean combinations of simple constraints of the form $x \bowtie c$ with $x \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathcal{G}(X)$ and $\nu \in \mathbb{R}^X_{\geq 0}$, we write $\nu \models g$ when g holds according to ν.

Definition 1 *(Timed automata).* A *timed automaton* (TA) is a tuple $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, such that L is a finite set of *locations* with $l_0 \in L$ the *initial location*, X is a finite set of *clocks*, Σ is a finite set of *actions*, $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the *transition relation*. $F \subseteq L$ is a set of *accepting locations*.

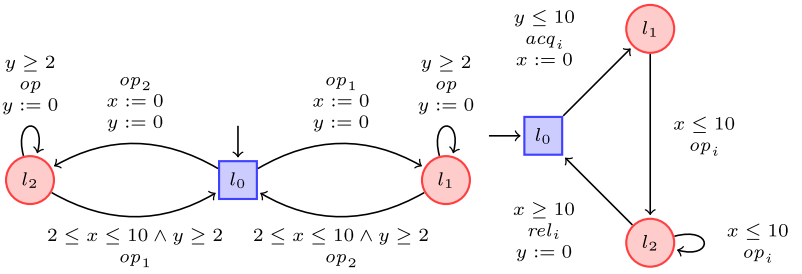
Example 1 *(Timed automata).* Let us consider again the specifications introduced in Section 2 where two processes access to and operate on a common resource. The global alphabet of events is $\Sigma \stackrel{\text{def}}{=} \{init, acq_1, rel_1, op_1, acq_2, rel_2, op_2, op\}$. The specifications on the behaviour of the processes introduced in Section 2 are formalised with the TAs in Fig. 6. Accepting locations are denoted by squares.

S_1 The specification is formalised by the automaton depicted in Fig. 6a with alphabet $\Sigma_1^i \stackrel{\text{def}}{=} \{rel_i, acq_i, op_i\}$ for process i, $i \in \{1,2\}$. The automaton has two clocks x and y, where clock x serves to keep track of the duration of the resource acquisition whereas clock y keeps track of the time elapsing between two operations. Both locations of the automaton are accepting and there are two implicit transitions from location l_1 to a trap state: i) upon action rel_i when the value of clock x is strictly lower than 10, and ii) upon action op_i when the value of clock y is strictly lower than 1.



(a) A safety automaton for S_1.

(b) A co-safety automaton for S_2.



(c) A regular automaton for S_3.

(d) A regular automaton for S_4. Note the implicit trap state reached from l_0 when $y > 10$ ensuring that there is no more than 10 t.u. without transaction.

Fig. 6. Some examples of timed automata.

S_2 The specification is formalised by the automaton depicted in Fig. 6b with alphabet $\Sigma_2 \stackrel{\text{def}}{=} \{init, op_1, op_2\}$. The automaton has two clocks, where clock x keeps track of the time elapsed since initialisation, whereas clock y keeps track of the time elapsing between the operations of the two different processes.

S_3 The specification is formalised by the automaton depicted in Fig. 6c with alphabet $\Sigma_3 \stackrel{\text{def}}{=} \{op, op_1, op_2\}$. Clock x keeps track of the time elapsing since the beginning of the transaction, whereas clock y keeps track of the time elapsing between any two operations.

S_4 The specification is formalised by the automaton depicted in Fig. 6d with alphabet $\Sigma_4^i \stackrel{\text{def}}{=} \{acq_i, op_i, rel_i\}$. Clock x keeps track of the duration of a currently executing transaction, whereas clock y keeps track of the time elapsing between two transactions.

The semantics of a TA is defined as follows.

Definition 2 *(Semantics of timed automata).* The *semantics* of a TA is a *timed transition system* $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ where $Q = L \times \mathbb{R}_{\geq 0}^X$ is the (infinite) set of *states*, $q_0 = (l_0, \nu_0)$ is the *initial state* where ν_0 is the valuation that maps every clock in X to 0, $Q_F = F \times \mathbb{R}_{\geq 0}^X$ is the set of *accepting states*, $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ is the set of transition *labels*, i.e., pairs composed of a delay and an action. The *transition relation* $\rightarrow \subseteq Q \times \Gamma \times Q$ is a set of transitions of the form $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ with $\nu' = (\nu + \delta)[Y \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ such that $\nu + \delta \models g$ for $\delta \in \mathbb{R}_{\geq 0}$.

In the following, we consider a timed automaton $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ with its semantics $[\![\mathcal{A}]\!]$. $\mathcal{A}$ is said to be *deterministic* whenever for any location l and any two distinct transitions (l, g_1, a, Y_1, l_1') and (l, g_2, a, Y_2, l_2') with source l and same action a in Δ, the conjunction of guards g_1 and g_2 is unsatisfiable. $\mathcal{A}$ is said to be *complete* whenever for any location $l \in L$ and any action $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labelled by a is valid. In the remainder of this paper, we shall consider only deterministic and complete timed automata, and, automata refer to timed automata.

Remark 1 *(Completeness and determinism).* Although we restrict the presentation to deterministic TAs, results may easily be extended to non-deterministic TAs, with slight adaptations required to the vocabulary and when synthesising an enforcement monitor. Regarding completeness, for readability of TA examples, if no transition can be triggered upon the reception

of an event, a TA implicitly moves to a non-accepting trap location (i.e., where all actions are looping with no timing constraint).

Remark 2 *(Other definitions of timed automata).* The definition of timed automata used in this paper is as the initial (and general) one proposed in [11] except that we do not use the Büchi acceptance condition because we deal with finite words. Even though we restrict constants in guards to be integers, and will see in Section 7 that TAs with rational constants may be necessary in the computation, those TAs can be transformed into integral TAs. Other definitions of timed automata have been proposed (see e.g., [12] for details). For instance, timed safety automata [13] are a simplified version of the original timed automata where invariants on locations replace the Büchi condition, as used in UPPAAL [14]. Several classes of determinisable automata with restrictions on the resets of clocks have been proposed. Event-recording (resp. event-predicting) timed automata [15] are timed automata with a clock associated to each action that records (resp. predicts) the time elapsed since the last occurrence (resp. the time of the next occurrence) of that action; event-clock automata have event-recording and event-predicting clocks.

A *run* ρ of $\mathcal{A}$ from a state $q \in Q$ is a sequence of moves in $[\![\mathcal{A}]\!]$: $\rho = q \xrightarrow{(\delta_1,a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n,a_n)} q_n$, for some $n \in \mathbb{N}$. The set of runs from the initial state $q_0 \in Q$ is denoted $\mathrm{Run}(\mathcal{A})$ and $\mathrm{Run}_{Q_F}(\mathcal{A})$ denotes the subset of those runs starting in q_0 and *accepted* by $\mathcal{A}$, i.e., ending in an accepting state $q_n \in Q_F$.

The *trace* started at date t of the run ρ is the timed word $(t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$ where $\forall i \in [1, n] : t_i = t + \sum_{j=1}^{i} \delta_j$ (the date of a_i is the sum of delays of the i first events plus t). We note $q \xrightarrow{w}_t q_n$ in this case, and generalise to $q \xrightarrow{w}_t P$ when $q_n \in P$ for a subset P of Q. We note $\xrightarrow{w}$ for $\xrightarrow{w}_0$. We note $\mathcal{L}(\mathcal{A})$ the set of traces started at date 0 of $\mathrm{Run}(\mathcal{A})$. We extend this notation to $\mathcal{L}_{Q_F}(\mathcal{A})$ as the set of traces of runs in $\mathrm{Run}_{Q_F}(\mathcal{A})$. We thus say that a timed word is accepted by $\mathcal{A}$ if it is the trace started at date 0 of an accepted run.

Example 2 *(Runs and traces of a timed automaton).* Consider the automaton in Fig. 6a. A possible *run* of this automaton from the initial state $(l_0, 0, 0)$ is the sequence of moves $(l_0, 0, 0) \xrightarrow{(1,acq_1)} (l_1, 0, 0) \xrightarrow{(2,op_1)} (l_0, 2, 0) \xrightarrow{(1,op_1)} (l_1, 3, 0) \xrightarrow{(0.5,acq_1)} (l_1, 3.5, 0.5) \xrightarrow{(0.5,op_1)} (l_1, 4, 0)$. The trace starting at date 0 of this run is the timed word $w_t = (1, acq_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (4.5, acq_1) \cdot (5, op_1)$. We have $(l_0, 0, 0) \xrightarrow{w_t} (l_1, 4, 0)$.

We now introduce the product of timed automata which is useful to intersect languages recognised by timed automata.

Definition 3 *(Product of timed automata).* Given two TAs $\mathcal{A}_1 = (L_1, l_1^0, X_1, \Sigma, \Delta_1, F_1)$ and $\mathcal{A}_2 = (L_2, l_2^0, X_2, \Sigma, \Delta_2, F_2)$ with disjoint sets of clocks, their product is the TA $\mathcal{A}_1 \times \mathcal{A}_2 \stackrel{\text{def}}{=} (L, l_0, X, \Sigma, \Delta, F)$ where $L = L_1 \times L_2$, $l_0 = (l_0^1, l_0^2)$, $X = X_1 \cup X_2$, $F = F_1 \times F_2$, and $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the *transition relation*, with $((l_1, l_2), g_1 \wedge g_2, a, Y_1 \cup Y_2, (l_1', l_2')) \in \Delta$ if $(l_1, g_1, a, Y_1, l_1') \in \Delta_1$ and $(l_2, g_2, a, Y_2, l_2') \in \Delta_2$.

It is easy to check that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

4.2. Partition of states of $[\![\mathcal{A}]\!]$

Given a TA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, with semantics $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \to, Q_F)$, the set of states Q of $[\![\mathcal{A}]\!]$ can be partitioned into four subsets *good* (G), *currently good* (G^C), *currently bad* (B^C) and *bad* (B), based on whether a state is accepting or not, and whether accepting or non-accepting states are reachable or not.

Formally, Q is partitioned into $Q = G^C \cup G \cup B^C \cup B$ where $Q_F = G^C \cup G$ and $Q \setminus Q_F = B^C \cup B$ and

- $G^C = Q_F \cap \mathrm{pre}^*(Q \setminus Q_F)$ i.e., the set of *currently good* states is the subset of accepting states from which non-accepting states are reachable,
- $G = Q_F \setminus G^C = Q_F \setminus \mathrm{pre}^*(Q \setminus Q_F)$ i.e., the set of *good* states is the subset of accepting states from which only accepting states are reachable,
- $B^C = (Q \setminus Q_F) \cap \mathrm{pre}^*(Q_F)$ i.e., the set of *currently bad* states is the subset of non-accepting states from which accepting states are reachable,
- $B = (Q \setminus Q_F) \setminus \mathrm{pre}^*(Q_F)$ i.e., the set of *bad* states is the subset of non-accepting states from which only non-accepting states are reachable,

where, for a subset P of Q, $\mathrm{pre}^*(P)$ denotes the set of states from which the set P is reachable.

It is well known that reachability of a set of locations is decidable using the classical zone (or region) symbolic representation (see [16]) and is PSPACE-complete. Since Q_F is the set of states with location F, this result can be used to compute the partition of Q.

By definition, from good (resp. bad) states, one can only reach good (resp. bad) states. Consequently, a run of a TA traverses currently good and/or currently bad states, and may eventually reach a good state and remain in good states, or a bad state and remain in bad states, or in pathological cases, it can directly start in good or bad states. This partition will be useful to characterise the classes of safety and co-safety timed properties, as explained in Section 4.3, and later for the synthesis of enforcement mechanisms.

4.3. Some sub-classes of regular timed properties

Regular, safety, and co-safety timed properties In this paper, a timed property is defined by a timed language $\varphi \subseteq \text{tw}(\Sigma)$ that can be recognised by a timed automaton. That is, we consider the set of regular timed properties. Given a timed word $\sigma \in \text{tw}(\Sigma)$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$. Safety (resp. co-safety) properties are sub-classes of regular timed properties. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). In this paper, the classes are characterised as follows:

Definition 4 *(Regular, safety, and co-safety properties).* We consider the following three classes of timed properties.

- Regular properties are the properties that can be defined by languages accepted by a TA.
- Safety properties are the non-empty prefix-closed timed languages that can be accepted by a TA.
- Co-safety properties are the non-universal[3] extension-closed timed languages that can be accepted by a TA.

The sets of safety and co-safety properties are subsets of the set of regular properties.

Safety and co-safety timed automata In the sequel, we shall only consider the properties that can be defined by deterministic and complete timed automata (Definition 1). Note that some of these properties can be defined using a timed temporal logic such as a subclass of MTL, which can be transformed into timed automata using the technique described in [17,18].

We now define syntactic restrictions on TAs that guarantee that a regular property defined by a TA defines a safety or a co-safety property.

Definition 5 *(Safety and co-safety TA).* Let $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ be a complete and deterministic TA, where $F \subseteq L$ is the set of accepting locations. $\mathcal{A}$ is said to be:

- a *safety* TA if $l_0 \in F \wedge \nexists (l, g, a, Y, l') \in \Delta : l \in L \setminus F \wedge l' \in F$;
- a *co-safety* TA if $l_0 \notin F \wedge \nexists (l, g, a, Y, l') \in \Delta : l \in F \wedge l' \in L \setminus F$.

It is then easy to check that safety (respectively co-safety) TAs define safety (respectively co-safety) properties.[4] Intuitively, a safety TA starts in the accepting location l_0 and has no transition from non-accepting to accepting locations. Thus, either all reachable locations are accepting (in this case, the TA recognises the universal language since it is complete), or the TA stays in accepting locations before possibly jumping definitively to non-accepting locations. At the semantic level a safety TA either has only good states (case of the universal language), or its runs start in the set of currently good states and may definitively jump in either the set of bad or the set of good states (no currently bad state can be reached). Thus, a safety TA defines a prefix-closed language. Conversely, a co-safety TA starts in the non-accepting location l_0 and has no transition from accepting to non-accepting locations. Thus, either all reachable locations are non-accepting (in this case, the TA recognises the empty language), or it stays in non-accepting locations before possibly jumping definitively to accepting locations. At the semantic level, a co-safety TA either only has bad states (case of the empty language), or its runs start in the set of currently bad states and may definitively jump in either the set of good states or the set of bad states (currently good state cannot be reached). Thus, a co-safety TA defines an extension-closed language.

Example 3 *(Classes of timed automata).* Let us consider again the specifications introduced in Example 6. We formalise specification S_i as property φ_i, $i = 1, \ldots, 4$. Property φ_1 is a safety property specified by the safety TA in Fig. 6a (leaving accepting locations is definitive). Property φ_2 is a co-safety property specified by the co-safety TA in Fig. 6b (leaving non-accepting locations is definitive). Property φ_3 is specified by the TA in Fig. 6c. Property φ_4 is specified by the TA in Fig. 6d. Both properties φ_3 and φ_4 are regular, but neither safety nor co-safety properties. In the underlying automata, runs may alternate between accepting and non-accepting locations, thus the languages that they define are neither prefix nor extension-closed.

[3] The universal property over $\mathbb{R}_{\geq 0} \times \Sigma$ is $\text{tw}(\Sigma)$.

[4] As one can observe, these definitions of safety and co-safety TAs slightly differ from the usual ones by expressing constraints on the initial state. As a consequence of these constraints, consistently with Definition 4, the empty and universal properties are ruled out from the set of safety and co-safety properties, respectively.

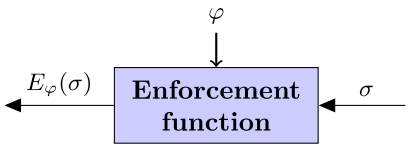


Fig. 7. Enforcement function.

5. Enforcement monitoring in a timed context

We now introduce our enforcement monitoring framework (Section 5.1) and specify the expected constraints on the input/output behaviour of enforcement mechanisms (Section 5.2).

5.1. General principles

To ease the design and implementation of enforcement monitoring mechanisms in a timed context, we describe enforcement mechanisms at three levels of abstraction: *enforcement functions*, *enforcement monitors*, and *enforcement algorithms*. An enforcement function describes the transformation of an input timed word into an output timed word at an abstract level where the whole input timed word is considered. In this section, we first formalise the constraints enforcement functions must satisfy, which reflect both physical constraints related to time, and required properties relating the input to the output. In Section 6, we shall define such enforcement functions, and prove that they satisfy the constraints. An enforcement monitor is a more concrete view and defines the operational behaviour of the enforcement mechanism over time. In Section 7, we shall define enforcement monitors as extended transition systems and we prove that, for a given property φ, the associated enforcement monitor implements the corresponding enforcement function. In other words, an enforcement function serves as an abstract description (black-box view) of an enforcement monitor, and an enforcement monitor is the operational description of an enforcement function. An enforcement algorithm (see Section 8) is an implementation of an enforcement monitor.

5.2. Constraints on an enforcement mechanism

At an abstract level, an enforcement mechanism for a given property φ can be seen as a function which takes as input a timed word and outputs a timed word. At this level, the input is considered as a whole, and the output is the corresponding whole timed word eventually produced, after an unbounded time elapse. In other words, the delay to observe the input and to produce the output is not considered. This is schematised in Fig. 7 and defined in Definition 6.

Definition 6 *(Enforcement function signature).* For a timed property φ, an enforcement mechanism behaves as a function, called *enforcement function* $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$.

An enforcement function E_φ models a mechanism that reads some input timed word σ from an event emitter, which is possibly incorrect w.r.t. φ, and transforms it into a timed word that satisfies φ which is output to the event receiver.

Before providing the actual definition of enforcement function in Section 6, we define the constraints that should be satisfied by an enforcement mechanism. The following constraints can serve as a specification of the expected behaviour of enforcement mechanisms for timed properties, that can delay and suppress events.

An enforcement mechanism should first satisfy some *physical constraint* reflecting the streaming of events: the output stream can only be modified by appending new events to its tail. Second, it should be *sound* w.r.t. the monitored property, meaning that it should correct input words according to φ if possible, and otherwise produce an empty output. Third, it should be *transparent*, which means that it is only allowed to shift events in time while keeping their order (we refer to such kind of mechanisms as time retardants) and suppress some events. These constraints are formalised in the following definition:

Definition 7 *(Constraints on an enforcement mechanism).* Given a timed property φ, an enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, should satisfy the following constraints:

- **Physical constraint:**

$$\forall \sigma, \sigma' \in \mathrm{tw}(\Sigma) : \sigma \preccurlyeq \sigma' \implies E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma') \qquad (\textbf{Phy}).$$

- **Soundness:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon \qquad (\textbf{Snd}).$$

– **Transparency:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \lhd_d \sigma \qquad (\mathbf{Tr}).$$

The physical constraint **(Phy)** means that E_φ is monotonic: the output produced for an extension σ' of an input word σ extends the output produced for σ. This stems from the fact that, over time, what is actually output by the enforcement function is a continuously growing timed word, i.e., what is output for a given input timed word can only be modified by appending new events with greater dates. Soundness **(Snd)** means that the output either satisfies property φ, or is empty. This allows to output nothing if there is no way to satisfy φ. Note that, together with the physical constraint, soundness implies that no event can be appended to the output before being sure that the property will be eventually satisfied with subsequent output events. Transparency **(Tr)** means that the output is a delayed subsequence of the input σ, thus the enforcement function is allowed to either suppress input events, or increase their dates while preserving their order.

It can be easily checked on the examples in Section 2 that the output sequences satisfy the constraints of enforcement mechanisms.

Remark 3. The soundness, transparency, and physical constraints describe the expected input/output behaviour of an enforcement function for the whole input sequence. Note that it however does not strongly constrain the output. In particular, an enforcement function that never produces any output complies to these constraints. However, to be practical, an actual enforcement function should also provide some guarantees on the output sequence it produces in terms of length and delay w.r.t. the input sequence. Such guarantees are specified by an optimality property in Section 6.

6. Enforcement functions: input/output description of enforcement mechanisms

We now define an enforcement function dedicated to a desired property φ. Its purpose is to define, at an abstract level, for any input word σ, the output word $E_\varphi(\sigma)$ expected from an enforcement mechanism that works as a delayer with suppression, where suppression only happens upon the reception of an event that prevents any satisfaction of the property in the future.

First, we discuss some preliminaries (Section 6.1) regarding the consequences of the choice of suppression strategy on efficiency and on the possible output sequences of the enforcement function. Then, we define the enforcement function itself, and prove in Section 6.2 that this functional definition satisfies the physical, soundness, and transparency constraints. We also prove that the enforcement function satisfies some optimality criterion with the chosen suppression strategy. Finally, in Section 6.3, we explain how the enforcement function behaves over time (how a given input sequence is consumed over time, and how the output is released in an incremental fashion).

6.1. Preliminaries to the definition of enforcement functions

An enforcement mechanism needs to memorise events since, for some properties (typically co-safety properties), upon the reception of some input timed word, the property might not be yet satisfiable by delaying, but a continuation of the input may allow satisfaction. For more general properties (which are neither safety nor co-safety properties), there may exist some prefix for which the property is satisfiable by delaying the input, thus dates can be chosen for these events. For efficiency reasons, and for our enforcement mechanisms to be amenable to online implementations, we also want to build the output in a fashion that is as incremental as possible. Enforcement mechanisms should thus take decisions on dates of output events as soon as possible. Still for efficiency considerations, we impose that suppression should occur only when necessary, i.e., when, upon the reception of a new event, there is no possibility to satisfy the property, whatever is the continuation of the input. Moreover, we decide to suppress the last received event only because it causes the unsatisfiability (even) if delayed. Note, when an enforcement mechanism decides not to suppress an action, it should not modify its decision in the future. Our choice of suppression strategy mainly stems from efficiency reasons. We discuss our choice and possible alternatives in Remark 4 in Section 6.2 (p. 15).

6.2. Definition of enforcement functions

As intuitively explained in the motivating examples of Section 2, during the correction of an input timed word σ, some subsequence of events σ_c is temporarily stored, until some new event (t, a) eventually allows to satisfy the property for the first time, or satisfy it again, by delaying the sequence $\sigma'_c = \sigma_c \cdot (t, a)$. For such a sequence σ'_c, the definition of an enforcement function shall use the set $\mathrm{CanD}(\sigma'_c)$ of candidate delayed sequences of σ'_c, independently of the property φ.

$$\mathrm{CanD}(\sigma'_c) = \left\{ w \in \mathrm{tw}(\Sigma) \mid w \succcurlyeq_d \sigma'_c \wedge \mathrm{start}(w) \geq \mathrm{end}(\sigma'_c) \right\}.$$

The set $\mathrm{CanD}(\sigma'_c)$ is the set of timed words w that delay σ'_c, and start at or after the ending date of σ'_c (which is the date t of the last event (t, a) of σ'_c). As we shall see, $w \succcurlyeq_d \sigma'_c$ stems from the fact that we consider enforcement mechanisms as time retardants, while $\mathrm{start}(w) \geq \mathrm{end}(\sigma'_c)$ means that the eligible timed words should not start before the date of its

last event (which is the current date t), as illustrated informally with specification S_3 in Section 2 and further discussed in Section 6.3.

With this preliminary notation, the enforcement function for a property φ can be defined as follows:

Definition 8 *(Enforcement function).* The enforcement function for a property φ is the function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ defined as:

$$E_\varphi(\sigma) = \Pi_1\left(\mathrm{store}_\varphi(\sigma)\right),$$

where $\mathrm{store}_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma) \times \mathrm{tw}(\Sigma)$ is defined as

$$\begin{aligned}
\mathrm{store}_\varphi(\epsilon) &= (\epsilon, \epsilon)\\
\mathrm{store}_\varphi(\sigma \cdot (t,a)) &= \begin{cases} (\sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}}\left(\kappa_\varphi(\sigma_s,\sigma_c')\right), \epsilon) & \text{if } \kappa_\varphi(\sigma_s,\sigma_c') \neq \emptyset,\\ (\sigma_s,\sigma_c) & \text{if } \kappa_{\mathrm{pref}(\varphi)}(\sigma_s,\sigma_c') = \emptyset,\\ (\sigma_s,\sigma_c') & \text{otherwise,}\end{cases}\\
&\quad \text{with } \sigma \in \mathrm{tw}(\Sigma), t \in \mathbb{R}_{\geq 0}, a \in \Sigma,\\
&\quad (\sigma_s,\sigma_c) = \mathrm{store}_\varphi(\sigma), \text{ and } \sigma_c' = \sigma_c \cdot (t,a),
\end{aligned}$$

where, for $\mathcal{L} \subseteq \mathrm{tw}(\Sigma)$,

$$\kappa_{\mathcal{L}}(\sigma_s,\sigma_c') \stackrel{\mathrm{def}}{=} \mathrm{CanD}(\sigma_c') \cap \sigma_s^{-1} \cdot \mathcal{L}.$$

For a given input σ, function store_φ computes a pair (σ_s, σ_c) of timed words: σ_s, which is extracted by the projection function Π_1 to produce the output $E_\varphi(\sigma)$; σ_c is used as a temporary memory. The pair (σ_s, σ_c) should be understood as follows:

- σ_s is a delayed subsequence of the input σ, in fact of its prefix of maximal length for which the absolute dates can be computed to satisfy property φ;
- σ_c is a subsequence of the remaining suffix of σ for which the releasing dates of events, still have to be computed. It is a subsequence (and not the complete suffix) since some events may have been suppressed when no delaying allowed to satisfy φ, whatever is the continuation of σ, if any.

Function E_φ incrementally computes a timed word according to the input timed word, and is defined inductively as follows. When the empty word ϵ is input, it produces (ϵ, ϵ). Otherwise, suppose that for the input σ the result of $\mathrm{store}_\varphi(\sigma)$ is (σ_s, σ_c) and consider a new received event (t, a). Now, the new timed word to correct is $\sigma_c' = \sigma_c \cdot (t, a)$. There are three possible cases, according to the vacuity of the two sets $\kappa_\varphi(\sigma_s, \sigma_c')$ and $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c')$. These sets are obtained respectively as the intersection of the set $\mathrm{CanD}(\sigma_c')$ with $\sigma_s^{-1} \cdot \varphi$ and $\sigma_s^{-1} \cdot \mathrm{pref}(\varphi)$. Let us recall that:

- $\mathrm{CanD}(\sigma_c')$ is the set of timed words that delay σ_c', and start at or after the ending date of σ_c' (i.e., the date of its last event (t, a)), since choosing an earlier date would cause the date to be already elapsed before the event could be released as output;
- $\sigma_s^{-1} \cdot \varphi = \{w \in \mathrm{tw}(\Sigma) \mid \sigma_s \cdot w \models \varphi\}$ is the set of timed words that satisfy φ after reading σ_s; similarly, since $\mathrm{pref}(\varphi) = \{v \in \mathrm{tw}(\Sigma) \mid \exists w' \in \mathrm{tw}(\Sigma) : v \cdot w' \models \varphi\}$ we get that $\sigma_s^{-1} \cdot \mathrm{pref}(\varphi) = \left\{w \in \mathrm{tw}(\Sigma) \mid \exists w' \in \mathrm{tw}(\Sigma) : \sigma_s \cdot w \cdot w' \models \varphi\right\}$, and thus $\sigma_s^{-1} \cdot \mathrm{pref}(\varphi)$ is the set of timed words for which a continuation satisfies φ after reading σ_s.

Thus $\kappa_\varphi(\sigma_s, \sigma_c')$ is the set of timed words w that belong to the candidate delayed sequences of σ_c' and such that $\sigma_s \cdot w$ satisfies φ; and $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c')$ is the set of timed words w that belong to the candidate delayed sequences of σ_c', and such that some additional continuation w' may satisfy φ, i.e., $\sigma_s \cdot w \cdot w' \models \varphi$. Note that, since $\kappa_\varphi(\sigma_s, \sigma_c') \subseteq \kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c')$, we distinguish three different cases:

- If $\kappa_\varphi(\sigma_s, \sigma_c') \neq \emptyset$ (and thus $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c') \neq \emptyset$), it is possible to choose appropriate dates for the timed word $\sigma_c' = \sigma_c \cdot (t, a)$ to satisfy φ. The minimal timed word in $\kappa_\varphi(\sigma_s, \sigma_c')$ w.r.t. the lexicographic order is chosen among those with minimal ending date, and appended to σ_s; the second element of the pair is set to ϵ since all events memorised in $\sigma_c \cdot (t, a)$ are corrected and appended to σ_s.
- If $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c') = \emptyset$ (and thus $\kappa_\varphi(\sigma_s, \sigma_c') = \emptyset$), it means that, whatever is the continuation of the current input $\sigma \cdot (t, a)$, there is no chance to find a correct delaying for (t, a). Thus, event (t, a) should be suppressed, leaving σ_c and σ_s unmodified.
- Otherwise, i.e., when $\kappa_\varphi(\sigma_s, \sigma_c') = \emptyset$ but $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c') \neq \emptyset$, it means that it is not yet possible to choose appropriate dates for $\sigma_c' = \sigma_c \cdot (t, a)$ to satisfy φ, but there is still a chance to do it in the future, depending on the continuation of the input, if any. Thus σ_c is modified into $\sigma_c' = \sigma_c \cdot (t, a)$ in memory, but σ_s is left unmodified.

Remark 4 *(Alternative strategies to suppress events).* When there is no possibility to continue correcting the input sequence (i.e., $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t,a)) = \emptyset$), we choose to erase only the last received event (t,a), since it is the one that causes this impossibility. However, other policies to suppress events could be chosen. In fact, one could choose to suppress any events in $\sigma_c \cdot (t,a)$, since dates of these events have not yet been chosen. This would then require to choose among all such subsequences, using an appropriate order. This may be rather complex to define, and, more importantly, computationally expensive because one would have to face the combinatorial explosion induced when considering the $2^{|\sigma_c \cdot (t,a)|}$ possible subsets of actions to suppress. Moreover, let us notice that enforcement mechanisms are purposed to work in an online fashion, hence making decisions on each reception of a new event. For this purpose, not reconsidering the suppression choices makes them definitive and lowers the computation related to suppression.

Proposition 1. *Given some property φ, its enforcement function E_φ as per Definition 8 satisfies the physical* (**Phy**)*, soundness* (**Snd**)*, and transparency* (**Tr**) *constraints as per Definition 7.*

Proof of Proposition 1 – sketch only. The proof of the physical constraint is a direct consequence of the definition of store_φ. The proofs of soundness and transparency follow the same pattern: they rely on an induction on the length of the input word σ. The induction steps use a case analysis, depending on whether the last input subsequence (i.e., the events in $\sigma_c \cdot (t,a)$) can be corrected or not. The complete proofs are given in Appendix A.1. □

In addition to the physical, soundness, and transparency constraints, the functional definition also ensures that each subsequence is output as soon as possible, as expressed by the following proposition.

Proposition 2 *(Optimality of enforcement functions). Given some property φ, its enforcement function E_φ as per Definition 8 satisfies the following optimality constraint:*

$$\begin{aligned}
&\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) = \epsilon \vee \exists m, w \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) = m \cdot w (\models \varphi),\ \textit{with}\\
&\quad m = \max^{\varphi}_{\prec,\epsilon}(E_\varphi(\sigma)),\ \textit{and}\\
&\quad w = \min_{\preceq_{\mathrm{lex}},\mathrm{end}}\{w' \in m^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))\\
&\qquad\qquad \wedge\, m \cdot w' \lhd_d \sigma \wedge \mathrm{start}(w') \geq \mathrm{end}(\sigma)\}
\end{aligned}$$

where $\max^{\varphi}_{\prec,\epsilon}(\sigma)$ is the maximal strict prefix of σ belonging to φ, formally:

$$\max^{\varphi}_{\prec,\epsilon}(\sigma) \stackrel{\mathrm{def}}{=} \max_{\preceq}\left(\left\{\sigma' \in \varphi \mid \sigma' \prec \sigma\right\} \cup \{\epsilon\}\right).$$

For any input σ, if the output $E_\varphi(\sigma)$ is not empty, then (it satisfies φ by soundness and) the output can be separated into a prefix m which is the maximal strict prefix of $E_\varphi(\sigma)$ satisfying property φ, and a suffix w. The optimality condition focuses on this last part, which is the suffix that allows to satisfy (again) the property. However, since the property considers any input σ, the same holds for every prefix of the input that allows to satisfy φ by enforcement, thus for any such (temporary) last subsequence.

The optimality constraint expresses that, among those sequences w' that could have been chosen (see below), w is the minimal one in terms of ending date, and lexical order (this second minimality ensures uniqueness). The "sequences that could have been chosen" are those such that $m \cdot w'$ satisfies the property, have the same events (thus can be produced by suppressing the same events), are delayed subsequences of the input σ, and have a starting date greater than or equal to $\mathrm{end}(\sigma)$, since $\mathrm{end}(\sigma)$ is the date at which w' is appended to the output, and thus a smaller date would be in the past of the output event.

Proof of Proposition 2 – sketch only. The proofs rely on an induction on the length of the input word σ. The induction step uses a case analysis, depending on whether the last input subsequence (i.e., the events in $\sigma_c \cdot (t,a)$) can be corrected or not. The proof is given in Appendix A.2 (p. 32). □

Remark 5 *(On the optimality condition).* Note that the condition $\Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))$ in Proposition 2 stems from the strategy chosen to suppress events (see Remark 4). If an enforcement function is defined, such that it is allowed to suppress any event in $\sigma_c \cdot (t,a)$, then the condition $\Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))$ in optimality should be removed. Moreover, note that optimality has to be defined in a recursive manner. Indeed, since enforcement mechanisms should produce output sequences in an incremental fashion, the optimal output that should be produced for an input sequence depends on the optimal outputs that have been produced for the prefixes of the input sequence. Because of the performance reasons mentioned in Remark 4, defining a more general notion of optimality (possibly parameterised by the suppression strategy) is left for future work.

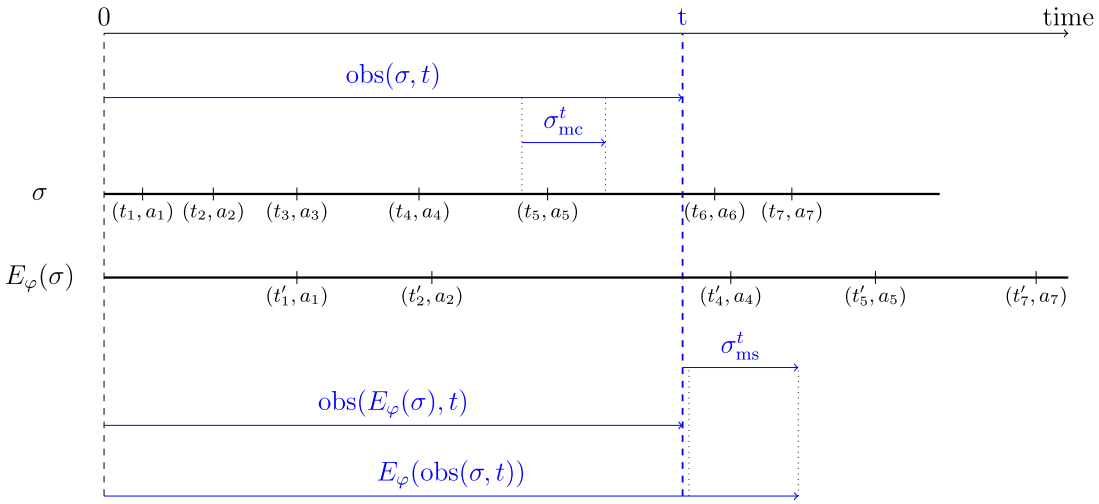


Fig. 8. Behaviour of enforcement functions over time.

6.3. Behaviour of enforcement functions over time

At an abstract level, an enforcement function takes as input a timed word and computes as output the timed word that is eventually produced by the enforcement mechanism after some unbounded delay. However, at a more concrete level, enforcement obeys some temporal constraints relative to the current date t. Firstly, since the enforcement mechanism reads the input timed word σ as a stream, what it can effectively observe from σ at date t is its prefix $\text{obs}(\sigma, t)$. Consequently, at date t, what it can compute from this observation is $E_\varphi(\text{obs}(\sigma, t))$. Note that it is legal to do so since, by definition $\text{obs}(\sigma, t)$ is a prefix of σ, and thus, by the physical constraint **(Phy)**, $E_\varphi(\text{obs}(\sigma, t))$ is a prefix of the complete output $E_\varphi(\sigma)$. Now, $E_\varphi(\text{obs}(\sigma, t))$ is a timed word where dates attached to events model the date when they should eventually be released as output. But at date t, only its prefix $\text{obs}(E_\varphi(\text{obs}(\sigma, t)), t)$ is effectively released as output. Now, notice that, since E_φ behaves as a time retardant (i.e., dates attached to output events exceed dates of corresponding input events), and $E_\varphi(\text{obs}(\sigma, t))$ is a prefix of $E_\varphi(\sigma)$, we get $\text{obs}(E_\varphi(\text{obs}(\sigma, t)), t) = \text{obs}(E_\varphi(\sigma), t)$. From this, we conclude that the released output at date t is $\text{obs}(E_\varphi(\sigma), t)$. Finally, what is ready to be released at date t, but not yet released is the residual of $E_\varphi(\text{obs}(\sigma, t))$ after observing $\text{obs}(E_\varphi(\sigma), t)$ thus $\text{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\text{obs}(\sigma, t))$. The enforcement monitor described in the next section, which implements the enforcement function, takes care of this temporal behaviour.

Example 4 *(Behaviour of enforcement functions over time).* (See Fig. 8.) Let us consider the input timed word $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdot (t_3, a_3) \cdot (t_4, a_4) \cdot (t_5, a_5) \cdot (t_6, a_6) \cdot (t_7, a_7)$, and let the output of the enforcement function be $E_\varphi(\sigma) = (t'_1, a_1) \cdot (t'_2, a_2) \cdot (t'_4, a_4) \cdot (t'_5, a_5) \cdot (t'_7, a_7)$. At time instant t:

- the observation of σ is $\text{obs}(\sigma, t) = (t_1, a_1) \cdot (t_2, a_2) \cdot (t_3, a_3) \cdot (t_4, a_4) \cdot (t_5, a_5)$,
- the subsequence of remaining suffix of $\text{obs}(\sigma, t)$ for which the releasing dates still have to be computed is $\sigma^t_{\text{mc}} = (t_5, a_5)$,
- the output that the enforcement function can compute from $\text{obs}(\sigma, t)$ is $E_\varphi(\text{obs}(\sigma, t)) = (t'_1, a_1) \cdot (t'_2, a_2) \cdot (t'_4, a_4)$,
- the released output is $\text{obs}(E_\varphi(\text{obs}(\sigma, t)), t) = \text{obs}(E_\varphi(\sigma), t) = (t'_1, a_1) \cdot (t'_2, a_2)$;
- the timed word ready to be released, denoted by σ^t_{ms}, is $\text{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\text{obs}(\sigma, t)) = (t'_4, a_4)$.

Example 5 *(Enforcement function).* We illustrate how Definition 8 is applied to enforce specification S_3 (see Section 2), formalised by property φ_3, recognised by the automaton depicted in Fig. 6c with Σ_3 $(= \{op_1, op_2, op\})$, and the input timed word $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. Fig. 9 shows the evolution of the observed input timed word $\text{obs}(\sigma_3, t)$, the output of the store_φ function when the input timed word is $\text{obs}(\sigma_3, t)$, and E_{φ_3}. Variable t keeps track of physical time, i.e., it contains the current date. When $t < 6$, the observed output is empty (since $E_{\varphi_3}(\text{obs}(\sigma_3, t)) = \epsilon$). When $t \geq 6$, the observed output, is $\text{obs}((6, op_1) \cdot (8, op) \cdot (10, op_2), t)$ (since $E_{\varphi_3}(\text{obs}(\sigma_3, t)) = (6, op_1) \cdot (8, op) \cdot (10, op_2)$).

Example 6 *(Enforcement function: a non-enforceable property).* Consider specification S_4 formalised by property φ_4, recognised by the automaton depicted in Fig. 6d with $\Sigma^i_4 = \{acq_i, op_i, rel_i\}$, and the input timed word $\sigma_4 = (3, acq_i) \cdot (7, op_i) \cdot (12, rel_i)$. Fig. 10 shows the evolution of the observation of the input timed word $\text{obs}(\sigma_4, t)$, the output of the store_φ function when the input timed word is $\text{obs}(\sigma_4, t)$, and E_{φ_4}. The output of the enforcement function is ϵ at any date because delaying action acq_i for 9 t.u. (i.e., until observing action rel_i) violates the timing constraint of 10 t.u. without transaction.

$t \in [0,2)$	$\begin{array}{l}\text{obs}\,(\sigma_3,t)=\epsilon\\ \text{store}_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))=(\epsilon,\epsilon)\\ \text{obs}\,(E_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))\,,t)=\text{obs}(\epsilon,t)=\epsilon\end{array}$
$t \in [2,3)$	$\begin{array}{l}\text{obs}\,(\sigma_3,t)=(2,op_1)\\ \text{store}_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))=(\epsilon,(2,op_1))\\ \text{obs}\,(E_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$
$t \in [3,3.5)$	$\begin{array}{l}\text{obs}\,(\sigma_3,t)=(2,op_1)\cdot(3,op_1)\\ \text{store}_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))=(\epsilon,(2,op_1))\\ \text{obs}\,(E_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$
$t \in [3.5,6)$	$\begin{array}{l}\text{obs}\,(\sigma_3,t)=(2,op_1)\cdot(3,op_1)\cdot(3.5,op)\\ \text{store}_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))=(\epsilon,(2,op_1)\cdot(3.5,op))\\ \text{obs}\,(E_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$
$t \in [6,\infty)$	$\begin{array}{l}\text{obs}\,(\sigma_3,t)=(2,op_1)\cdot(3,op_1)\cdot(3.5,op)\cdot(6,op_2)\\ \text{store}_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))=((6,op_1)\cdot(8,op)\cdot(10,op_2),\epsilon)\\ \text{obs}\,(E_{\varphi_3}\,(\text{obs}\,(\sigma_3,t))\,,t)=\text{obs}\,((6,op_1)\cdot(8,op)\cdot(10,op_2)\,,t)\end{array}$

Fig. 9. Evolution over time of the values of the enforcement function for property φ_3 specifying the transactional execution of actions op_1 and op_2.

$t \in [0,3)$	$\begin{array}{l}\text{obs}\,(\sigma_4,t)=\epsilon\\ \text{store}_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))=(\epsilon,\epsilon)\\ \text{obs}\,(E_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$
$t \in [3,7)$	$\begin{array}{l}\text{obs}\,(\sigma_4,t)=(3,acq_i)\\ \text{store}_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))=(\epsilon,(3,acq_i))\\ \text{obs}\,(E_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$
$t \in [7,12)$	$\begin{array}{l}\text{obs}\,(\sigma_4,t)=(3,acq_i)\cdot(7,op_i)\\ \text{store}_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))=(\epsilon,(3,acq_i)\cdot(7,op_i))\\ \text{obs}\,(E_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$
$t \in [12,\infty)$	$\begin{array}{l}\text{obs}\,(\sigma_4,t)=(3,acq_i)\cdot(7,op_i)\cdot(12,rel_i)\\ \text{store}_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))=(\epsilon,(3,acq_i)\cdot(7,op_i))\\ \text{obs}\,(E_{\varphi_4}\,(\text{obs}\,(\sigma_4,t))\,,t)=\text{obs}\,(\epsilon,t)=\epsilon\end{array}$

Fig. 10. Evolution over time of the values of the enforcement function for property φ_4 (a non-enforceable property).

Remark 6 *(Simplified enforcement functions for safety properties).* Because of the characteristics of safety properties, the enforcement function for such properties can be simplified. A safety property φ is prefix closed thus $\text{pref}(\varphi)=\varphi$, which implies that the two functions $\kappa_{\text{pref}(\varphi)}$ and κ_φ are identical. Thus, the two first cases in the definition of $\text{store}_\varphi(\sigma\cdot(t,a))$ can be simplified and distinguished according to whether $\kappa_\varphi(\sigma_s,\sigma_c')=\emptyset$ or not; and the third case never happens. Moreover, since σ_c is initially empty, and the two first cases in the definition of $\text{store}_\varphi(\sigma\cdot(t,a))$ do not modify σ_c, by a simple induction on the input sequence, we observe that σ_c always remains empty. Thus, the second output parameter of function store_φ (i.e., the internal memory) can be suppressed. Additionally, in the first case, the first argument of the output can be simplified as it is always called with the last read event (t,a) (see below).

The enforcement function for safety properties $\text{store}_\varphi^{\text{sa}}:\text{tw}(\Sigma)\to\text{tw}(\Sigma)$ can be defined as follows:

$$\begin{aligned}\text{store}_\varphi^{\text{sa}}(\epsilon) &= \epsilon\\ \text{store}_\varphi^{\text{sa}}(\sigma\cdot(t,a)) &= \begin{cases}\text{store}_\varphi^{\text{sa}}(\sigma)\cdot(\min(K(\sigma,(t,a))),a) & \text{if } K(\sigma,(t,a))\neq\emptyset,\\ \text{store}_\varphi^{\text{sa}}(\sigma) & \text{otherwise,}\end{cases}\end{aligned}$$

where $K(\sigma,(t,a))\stackrel{\text{def}}{=}\{t'\in\mathbb{R}_{\geq 0}\mid t'\geq t\wedge\text{store}_\varphi^{\text{sa}}(\sigma)\cdot(t',a)\lhd_d\sigma\cdot(t,a)\wedge\text{store}_\varphi^{\text{sa}}(\sigma)\cdot(t',a)\in\varphi\}$ is the set of dates $t'\geq t$ that can be associated to a such that the extension $\text{store}_\varphi^{\text{sa}}(\sigma)\cdot(t',a)$ of $\text{store}_\varphi^{\text{sa}}(\sigma)$ is a delayed subsequence of $\sigma\cdot(t,a)$ and still satisfies property φ.

7. Enforcement monitors: operational description of enforcement mechanisms

The enforcement function defined in Section 6 describes inductively how an input stream of events is transformed according to a property. It provides a functional view of enforcement mechanisms and could be implemented using functional programming constructs such as recursion and lazy evaluation.

However, a concern is that the computation of dates upon the reception of a new event also depends on the sequence of events σ_s that have been already corrected, through functions κ_φ and $\kappa_{\text{pref}(\varphi)}$. Consequently, implementing directly an enforcement function would require the enforcement mechanism to store σ_s in its memory, that would grow over time and never be emptied.

Instead, we implement an enforcement function E_φ for a property φ specified by a TA $\mathcal{A}_\varphi$ with an enforcement monitor (EM). An EM has an operational semantics: it is defined as a transition system $\mathcal{E}$, and has explicit state information. It keeps track of and uses information such as time elapsed, and the state reached after reading (or simulating) σ_s in the underlying TA to release the actions stored in σ_s at appropriate dates. Hence, an EM does not need to store σ_s.

7.1. Preliminaries to the definition of enforcement monitors

In contrast with an enforcement function which, at an abstract level, takes a timed word as input and produces a timed word as output, an enforcement monitor $\mathcal{E}$ also needs to take into account physical time (i.e., the actual date t), the current observation $\text{obs}(\sigma, t)$ of the input stream σ at date t, the release of events to the environment which is $\text{obs}(E_\varphi(\sigma), t)$, and the residual of $E_\varphi(\text{obs}(\sigma, t))$ after releasing $\text{obs}(E_\varphi(\sigma), t)$. Note, since storing these sequences would be impractical at runtime, an enforcement monitor encodes equivalent information as described below.

An EM $\mathcal{E}$ is equipped with: a clock which keeps track of the current date t; two memories and a set of enforcement operations used to store and release some timed events to and from the memories, respectively. The memories are basically queues, each of them containing a timed word:

- σ_{mc} manages the input queue, more precisely the subsequence of the input $\text{obs}(\sigma, t)$ consisting of non-suppressed events for which dates could not yet be chosen to satisfy the property. This exactly corresponds to the timed word σ_c in function store_φ (see Definition 8);
- σ_{ms} is the output queue which manages the part of the output $E_\varphi(\sigma)$ which is computed at date t but not yet released; since at date t only prefix $\text{obs}(\sigma, t)$ has been observed, and $\text{obs}(E_\varphi(\sigma), t)$ has already been released, σ_{ms} contains the residual $\text{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\text{obs}(\sigma, t))$, i.e., the timed word that is ready to be released but not yet released.

An EM also keeps track of the current state q of the underlying LTS of the TA $\mathcal{A}_\varphi$ that encodes φ and the date t_F at which q is reached. The current state q is the one reached after reading the timed word $E_\varphi(\text{obs}(\sigma, t))$ (that also corresponds to σ_s in the definition of E_φ), which is the output that can be computed from the current observation $\text{obs}(\sigma, t)$. By definition q is either q_0 or a state in Q_F. The date t_F is the date $\text{end}(E_\varphi(\text{obs}(\sigma, t)))$ (and evaluates to 0 if $E_\varphi(\text{obs}(\sigma, t)) = \epsilon$).

7.2. *Function* update

Before defining enforcement monitors, we introduce function update which takes as input the current state $q \in Q_F \cup \{q_0\}$ of $[\![\mathcal{A}_\varphi]\!]$[5] reached after reading $E_\varphi(\text{obs}(\sigma, t))$, the arrival date t_F in this state q, a timed word $\sigma_{\text{mc}} \in \text{tw}(\Sigma)$ that has to be corrected, and the last received event (t, a). Function update possibly updates the current state, and outputs a marker used by $\mathcal{E}$ to make decisions, according to whether σ_{mc} can be corrected or not, and in the negative case, whether an extension could be.

Definition 9 *(Function* update*).* update is a function from $Q \times \mathbb{R}_{\geq 0} \times \text{tw}(\Sigma) \times (\mathbb{R}_{\geq 0} \times \Sigma)$ to $Q \times \text{tw}(\Sigma) \times \{\texttt{ok}, \texttt{c\_bad}, \texttt{bad}\}$ defined as follows:

$$\text{update}(q, t_F, \sigma_{\text{mc}}, (t, a)) \stackrel{\text{def}}{=} \begin{cases} (q', w_{\min}, \texttt{ok}) & \text{if } k_{Q_F}(q, t_F, \sigma_{\text{mc}} \cdot (t, a)) \neq \emptyset \wedge q \overset{w_{\min}}{\rightarrow}_{t_F} q', \\ (q, \sigma_{\text{mc}}, \texttt{bad}) & \text{if } k_{Q_F \cup B^C}(q, t_F, \sigma_{\text{mc}} \cdot (t, a)) = \emptyset, \\ (q, \sigma_{\text{mc}} \cdot (t, a), \texttt{c\_bad}) & \text{otherwise,} \end{cases}$$

where, for $\mathcal{Q} \subseteq Q$,

$$k_{\mathcal{Q}}(q, t_F, \sigma) = \left\{ w \in \text{tw}(\Sigma) \mid q \overset{w}{\rightarrow}_{t_F} \mathcal{Q} \right\} \cap \text{CanD}(\sigma),$$

and $w_{\min} = \min_{\preceq_{\text{lex}}, \text{end}} k_{Q_F}(q, t_F, \sigma_{\text{mc}} \cdot (t, a))$.

Recall that $\text{CanD}(\sigma)$ (defined in section 6.2) computes the set of timed words that delay σ and start at or after $\text{end}(\sigma)$. Function $k_{\mathcal{Q}}$ explicitly uses the semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ of the TA $\mathcal{A}_\varphi$ defining property φ, and, using function CanD, mimics the computation of the sets $\kappa_\varphi(\sigma_s, \sigma_c')$, and $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma_c')$ defined in section 6.2. Function $k_{\mathcal{Q}}$ is parameterised with a set of states $\mathcal{Q} \subseteq Q$ and called with three parameters: the current state q, a date t_F, and a sequence σ. It returns the set of timed words leading to a state in $\mathcal{Q}$ from state q starting at date t_F, among sequences in $\text{CanD}(\sigma)$.

The three cases in the definition of update encode the three cases in the definition of function store_φ, in the same order:

[5] The partitioning of the states Q of $[\![\mathcal{A}]\!]$ into four subsets G, G^C, B^C and B is defined in Section 4.2.

- In the first case, $\mathcal{Q} = Q_F$ and $k_{Q_F}(q, t_F, \sigma_{\text{mc}} \cdot (t, a))$ is not empty, i.e., appropriate delaying dates can be chosen for the events in $\sigma_{\text{mc}} \cdot (t, a)$ such that an accepting state $q' \in Q_F$ is reached from q, starting at date t_F. In this case, function update returns q', $w_{\min}$, and marker `ok`: $w_{\min}$ is the minimal word w.r.t. the lexical order among those timed words of minimal ending date in $k_{Q_F}(q, t_F, \sigma_{\text{mc}} \cdot (t, a))$, $q' \in Q_F$ is the state reached from q with $w_{\min}$, and marker `ok` indicates that Q_F is reached.
- In the second case, $\mathcal{Q} = Q_F \cup B^C$ and $k_{Q_F \cup B^C}(q, t_F, \sigma_{\text{mc}} \cdot (t, a))$ is empty; it is thus impossible to correct $\sigma_{\text{mc}} \cdot (t, a)$ in the future, since no candidate sequence delaying $\sigma_{\text{mc}} \cdot (t, a)$ leads to a state in $Q_F \cup B^C$, i.e., accepting states or states from which a path leads to an accepting state (they all lead to bad states B). This reflects the fact that $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t, a))$ is empty, since the set of accepting states of pref(φ) is $Q_F \cup B^C$. In this case, function update returns state q and timed word σ_{mc} unmodified, indicating that event (t, a) is suppressed, and marker `bad` indicates that no accepting state could be reached in the future if (t, a) was retained in memory.
- In the third case, function update returns state q unmodified, but returns the timed word $\sigma_{\text{mc}} \cdot (t, a)$, and a marker `c_bad`. The marker indicates that $\sigma_{\text{mc}} \cdot (t, a)$ cannot be delayed to reach an accepting state, but there it is still possible to reach a new accepting state after observing more events in the future.

On the computation of function update Function update can be computed using operations on TAs and known algorithms solving classical problems, with the help of the standard symbolic representation of behaviours of TAs by region or zone graphs, and refinement of these, using Difference Bound Matrices (DBM) to encode timing constraints. However, one needs to adapt TAs following practical considerations as explained below. We first introduce the sub-problems involved in the computation of update and references to their algorithmic solutions. Next, we explain some considerations to extend the kind of TAs handled by these algorithms. Finally, we explain how to encode the computation of update into these algorithms and standard operations on TAs.

Reachability problem: For a TA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, check whether F is reachable. Recall that reachability is PSPACE-complete in the size of the TA $\mathcal{A}$ [11] and can be solved using the symbolic region or zone representations and forward or backward analysis, e.g., using UPPAAL [14].

Optimal reachability problem: For a TA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, check whether F is reachable and if yes, find a run with minimal duration. It can be proven that this problem is also PSPACE-complete in the size of $\mathcal{A}$. PSPACE-hardness is a direct consequence of the fact that reachability in TAs is already PSPACE-hard. PSPACE-easiness is a consequence of the fact that a more general problem, the optimal cost reachability problem for weighted timed automata (WTAs), is proven to be PSPACE-complete in [19], and can be solved by the exploration of the *weighted directed graph*. The weighted directed graph is a refinement of the region graph in which the durations of time transitions are arbitrarily close to integers, and edges are augmented with cost functions which are polynomials in the constants of $\mathcal{A}$. Cost-optimal paths can be found among those where the durations spent in locations are arbitrarily close to integers. Moreover, in the case of TAs with only non-strict guards, the optimal timed words indeed have integral dates.

We now state four considerations that allow to apply these algorithms in our context:

Consideration 1 In a real runtime environment, dates of input events are observed by a digital clock with limited precision. Observed dates can thus be considered as rationals, more precisely integral multiples of a sampling rate $1/D$ of a clock, rather than reals as in the idealised model of timed words. As a consequence of this, of the computation of update and its use in the enforcement monitor, the computed output dates are also rationals (obtained by reverse scaling of integer dates obtained by optimal reachability, see below).

Consideration 2 In our definition of TAs, all constants in guards are integers. These TAs are sometimes called *integral* TAs. As will be clear later, and in particular because of Consideration 1, we shall also consider *rational* TAs, i.e., TAs where constants can be rational. A rational TA can be transformed into an integral TA by considering $1/d$ as the new unit value, where d is the least common multiple of all denominators of rational constants. Note that the value $1/d$, which will be useful in the sequel, will always be a multiple of the observation sampling rate $1/D$, thus one can simply take $1/D$. Since the size of the regions/zones graph depends on the maximal constant, there is a tradeoff between the precision of the observation and the cost of reachability analysis.

Consideration 3 Still due to Consideration 1, in the use of update we will have to solve (optimal) reachability not only from the initial state, but from some state q where the location l may differ from l_0 and clocks have a rational valuation ν. First, as is the case with Consideration 2, one can scale this TA by multiplying all constants by the least common multiple of denominators of this valuation (and constants if rational) in order to get an integral TA starting in an integral valuation ν'. Second, the construction of the region/zone automaton will be as usual, except that it should start in (l, ν').

Consideration 4 As seen above with optimal reachability, for TAs with strict (lower) guards, infimum may not be realisable even though reachability is achieved. However, a timed word arbitrarily close to the infimum exists as soon as reachability is achieved. Alternatively, one may approximate the TA with a TA exhibiting only non-strict guards. Since output dates should be multiple of the sampling rate $1/D$, the approximation consists in transforming all strict guards of the

form $x > c$, where c is an integer, into guards of the form $x \geq c + 1/D$, and then use Consideration 2 to transform this rational TA into an integral TA, and get guards of the form $x \geq D * c + 1$.

Remark 7. Most of the above considerations concern rational constants (or initial rational valuations) in TAs and have their roots in the precision of the observation (Consideration 1). An alternative way to understand those considerations, and avoid problems of rational constants in all the algorithms, is simply to consider the observation sampling $1/D$ as the new time unit, and thus to observe events at integral dates and rescale TA $\mathcal{A}_\varphi$ according to this new time unit. As a consequence, all TAs that have to be built would be integral TAs.

We now come back to the computation of function update. First, $\text{CanD}(\sigma_{\text{mc}} \cdot (t, a))$ can be represented by a rational TA C with a new clock $y \notin X$ (that does not belong to the set of clocks of the TA $\mathcal{A}_\varphi$ of the property) initialised to 0, and $|\sigma_{\text{mc}} \cdot (t, a)|$ transitions in sequence, one transition per action in $\sigma_{\text{mc}} \cdot (t, a)$, the first transition with constraint $y \geq t$, the other ones with no timing constraint, no reset on any transition, and one accepting location in set F_C at the end, with no outgoing transition. Clearly, this automaton recognises timed words delaying $\sigma_{\text{mc}} \cdot (t, a)$ and starting after t. Since t is the date at which a is observed, by Consideration 1 we suppose that it is a rational, multiple of the observation sampling $1/D$, thus C is a rational TA. For technical reasons, in the following we will rather consider the rational TA C' obtained from C by replacing $y \geq t$ by $y \geq t - t_F$ in the first transition, where t_F is the arrival date in state q (note that it can be easily proven by induction on the use of update that t_F is rational, since computed output dates are rational). C' recognises the same timed words as C, with all dates decreased by the duration t_F.

For the first case in the definition of update, one needs first to check whether $k_{Q_F}(q, t_F, \sigma_{\text{mc}} \cdot (t, a)) \neq \emptyset$ and then to pick a timed word with minimal duration in this set. This can thus be done as follows: let $\mathcal{A}_\varphi(q)$ be the same TA as $\mathcal{A}_\varphi$, but starting in the initial state q, where q is a pair (l, ν) with $l \in L$ and ν a rational valuation of the clocks in X. Now build the product TA $\mathcal{A}_\varphi(q) \times C'$ and check whether $F \times F_C$ is reachable. For this purpose, Consideration 4 is used to transform strict guards into non-strict ones, and then Considerations 2 and 3 are used to transform this rational TA initialised with a rational valuation into an integral TA initialised with an integral valuation. If $F \times F_C$ is reachable, computing the timed word with minimum duration can be done using the algorithm described in [19], resulting in an integral timed word. Next, one has to rescale this integral timed word into a rational timed word by division by the scalings used to transform rational TAs to integral TAs. Finally, the resulting timed word is increased by the duration t_F to get the final result.

For the second case, one needs to check whether $k_{Q_F \cup B^C}(q, t_F, \sigma_{\text{mc}} \cdot (t, a)) = \emptyset$. This can be done as follows: let C'' be the same automaton as C', except that the accepting locations in F_C loop on any action; C'' then recognises extensions of timed words in $\text{CanD}(\sigma_{\text{mc}} \cdot (t, a))$, but again decreased by the duration t_F; build the product $\mathcal{A}_\varphi(q) \times C''$, and check whether $F \times F_C$ is reachable in this TA, using Considerations 2 and 3 again to transform this rational TA initialised with a rational valuation into an integral TA initialised with an integral valuation. If the answer is no, $k_{Q_F \cup B^C}(q, t_F, \sigma_{\text{mc}} \cdot (t, a)) = \emptyset$.

An operational definition of function update as an algorithm is described in Section 8.

Complexity Recall that for an integral timed automaton $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, reachability can be solved by first constructing the region graph or zone graph which size is in $\mathcal{O}((|\Delta| + |L|).(2M + 2)^{|X|}.|X|!.2^{|X|})$, where M is the maximal constant appearing in guards, $|L|$ is the number of locations, $|\Delta|$ the number of edges, and $|X|$ the number of clocks, and solving reachability in this finite graph. Since reachability in finite graphs is NLOGSPACE-complete (in the size of the finite graph), globally, this algorithm is in PSPACE, and it is proven that the problem is PSPACE-complete [11].

As previously mentioned, optimal reachability is PSPACE-complete in the size of $\mathcal{A}$. It is a consequence of the PSPACE-completeness of the more general problem of optimal reachability for weighted time automata. Weighted timed automata are extensions of timed automata where a cost function $\mathcal{C}$ assigns integer costs to both locations and transitions, with the semantics that firing a transition e induces a cost $\mathcal{C}(e)$, and spending τ time units in a location l induces a cost $\mathcal{C}(l).\tau$. The optimal reachability problem for TAs can thus be reduced to the cost optimal reachability problem for WTAs in which a null cost is assigned to transitions and a cost of 1 is assigned to every location. The optimal reachability problem is solved by first constructing the weighted directed graph, which refines the region graph by focusing on what happens close to integral corners of regions, and labelling transitions with a cost function. The size of the weighted directed graph is $|X| + 1$ times bigger than the region graph. Optimality is then solved by traversing on-the-fly this graph and comparing the weights of elementary paths.

For simplicity, the complexity of both algorithms are in general abstracted to $\mathcal{O}(2^{|\mathcal{A}|})$ where $|\mathcal{A}|$ takes into account the number of transitions, locations, the maximal constant, and the number of clocks in $\mathcal{A}$.

Note that to solve those problems for a rational TA, one first needs to build an integral TA according to the observation sampling $1/D$, by multiplying constants by D. The size of the region graph thus becomes $\mathcal{O}((|\Delta| + |L|).(2.D.M + 2)^{|X|}.|X|!.2^{|X|})$, and both problems are still in $\mathcal{O}(2^{|\mathcal{A}|})$. For the product of two TAs $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, and $\mathcal{B} = (L', l'_0, X', \Sigma, \Delta', F')$, with respective maximal constants M and M', the size of the region graph thus becomes $\mathcal{O}((|\Delta|.|\Delta'| + |L|.|L'|).(2.\max(M, M') + 2)^{|X|+|X'|}.(|X| + |X'|)!.2^{|X|+|X'|})$, and the complexity of (optimal) reachability becomes $\mathcal{O}(2^{|\mathcal{A}|+|\mathcal{B}|})$.

Now, let us come to the complexity of update, or more precisely to the orders of sizes of the region graphs and weighted directed graphs that need to be traversed, since these are the key elements in the complexity of the algorithms. For a given input memory $\sigma_{\text{mc}} \cdot (t, a)$, the computation of update requires to solve the optimal reachability problem on the automaton

$\mathcal{A}_\varphi(q) \times C'$ and the reachability problem on the automaton $\mathcal{A}_\varphi(q) \times C''$ where C' and C'' are built from $\sigma_{\text{mc}} \cdot (t, a)$ as explained above, and $\mathcal{A}_\varphi(q)$ is obtained from a TA $\mathcal{A}_\varphi = (L, l_0, X, \Sigma, \Delta, F)$ (with maximal constant M) by shifting the initial state to q. Firstly, note that the automaton $\mathcal{A}_\varphi(q)$ is of same size as $\mathcal{A}_\varphi$, but is a rational TA, thus the maximal constant of the corresponding integral automaton is $M.D$ when scaling to integral automata with observation sampling $1/D$. Secondly, the automata C' and C'' both have $\mathcal{O}(|\sigma_{\text{mc}}|)$ locations and respectively $\mathcal{O}(|\sigma_{\text{mc}}|)$ and $\mathcal{O}(|\sigma_{\text{mc}}| + |\Sigma|)$ transitions. They both have one clock and the maximal constant of their corresponding integral TAs is the integer $D.(t - t_F)$.

For the product TAs $\mathcal{A}_\varphi(q) \times C'$ and $\mathcal{A}_\varphi(q) \times C''$, we get $\mathcal{O}(|\sigma_{\text{mc}}|.|L|)$ locations and respectively $\mathcal{O}(|\sigma_{\text{mc}}|.|\Delta|)$ and $\mathcal{O}((|\sigma_{\text{mc}}| + |\Sigma|).|\Delta|)$ transitions. The maximal constant is $D.\max(M, t - t_F)$ and both automata have $|X| + 1$ clocks.

In the first case of function update, solving the optimal reachability problem in the TA $\mathcal{A}_\varphi(q) \times C'$ induces the (partial) construction and traversal of a weighted directed graph which is of size $\mathcal{O}((|X| + 2).|\sigma_{\text{mc}}|.(|\Delta| + |L|).(2.D.\max(M, t - t_F) + 2)^{|X|+1}.(|X| + 1)!.2^{|X|+1})$.

In the second case, the reachability problem in the TA $\mathcal{A}_\varphi(q) \times C''$ can be solved by building a region graph of size $\mathcal{O}(((|\sigma_{\text{mc}}| + |\Sigma|).|\Delta|) + (|\sigma_{\text{mc}}|.|L|)).(2.D.\max(M, t - t_F) + 2)^{|X|+1}.(|X| + 1)!.2^{|X|+1})$.

In spite of these complexities, these problems can be efficiently solved, e.g., in UPPAAL [14], using zones and their encoding with DBMs. One key to efficiency is the choice of the right observation sampling $1/D$ which influences the size of the maximal constant in integral TAs. The smaller is $1/D$, the tighter is the observation, but the larger is the region graph. It should also be noted that even though the maximal size of the product automata is in the product of sizes of component automata, in practice only paths in $\mathcal{A}_\varphi$ along the untimed projection of $\sigma_{\text{mc}} \cdot (t, a)$ have to be considered, which strongly restricts the region graph or weighted directed graph that need to be built when searching for (optimal) accepted timed words. Finally, as will be clear later, update is called with sequences $\sigma_{\text{mc}} \cdot (t, a)$ of increasing length (in the case where no suppression occurs), starting from 1, until it can be corrected to satisfy φ, in which case its length is reinitialised to 1. The worst case complexity is reached when no prefix can be corrected (but a possible extension always could) before the arrival of the sequence. But in general, we may expect that φ can be regularly satisfied by correcting the input.

7.3. Definition of enforcement monitors

We can now define the enforcement monitor using function update defined in Section 7.2.

Definition 10 *(Enforcement monitor).* Let us consider a regular property φ recognised by the TA $\mathcal{A}_\varphi$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Gamma, \to, Q_F)$. The enforcement monitor for φ is the transition system $\mathcal{E}_\varphi = (C^{\mathcal{E}_\varphi}, c_0^{\mathcal{E}_\varphi}, \Gamma^{\mathcal{E}_\varphi}, \hookrightarrow_{\mathcal{E}_\varphi})$ s.t.:

- $C^{\mathcal{E}_\varphi} = \text{tw}(\Sigma) \times \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \times Q \times \mathbb{R}_{\geq 0}$ is the set of configurations of the form $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F)$, where σ_{ms}, σ_{mc} are timed words to memorise events, t is a positive real number to keep track of time, q is a state in the semantics of the TA and t_F keeps track of the arrival date in q,
- $c_0^{\mathcal{E}_\varphi} = (\epsilon, \epsilon, 0, q_0, 0) \in C^{\mathcal{E}_\varphi}$ is the initial configuration,
- $\Gamma^{\mathcal{E}_\varphi} = \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ is the alphabet, i.e., the set of triples comprised of an optional input event, an operation, and an optional output event, where the set of possible operations is $Op = \{\text{store-}\overline{\varphi}(\cdot), \text{store}_{\text{sup}}\text{-}\overline{\varphi}(\cdot), \text{store-}\varphi(\cdot), \text{release}(\cdot), \text{idle}(\cdot)\}$;
- $\hookrightarrow_{\mathcal{E}_\varphi} \subseteq C^{\mathcal{E}_\varphi} \times \Gamma^{\mathcal{E}_\varphi} \times C^{\mathcal{E}_\varphi}$ is the transition relation defined as the smallest relation obtained by the following rules applied with the priority order below:
  - **1. *store-*φ**:
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F) \overset{(t,a)/\text{store}-\varphi(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}} \cdot w, \epsilon, t, q', \text{end}(w))$, if $\text{update}(q, t_F, \sigma_{\text{mc}}, (t, a))) = (q', w, \texttt{ok})$,
  - **2. *store*$_{\text{sup}}$*-*$\overline{\varphi}$**:
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F) \overset{(t,a)/\text{store}_{\text{sup}}-\overline{\varphi}(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F)$, if $\text{update}(q, t_F, \sigma_{\text{mc}}, (t, a)) = (q, \sigma_{\text{mc}}, \texttt{bad})$,
  - **3. *store-*$\overline{\varphi}$**:
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F) \overset{(t,a)/\text{store}-\overline{\varphi}(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}}, \sigma_{\text{mc}} \cdot (t, a), t, q, t_F)$, if $\text{update}(q, t_F, \sigma_{\text{mc}}, (t, a)) = (q, \sigma_{\text{mc}} \cdot (t, a), \texttt{c\_bad})$,
  - **4. *release***:
    $((t, a) \cdot \sigma'_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F) \overset{\epsilon/\text{release}(t,a)/(t,a)}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma'_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F)$,
  - **5. *idle***:
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F) \overset{\epsilon/\text{idle}(\delta)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t + \delta, q, t_F)$ if $\delta \in \mathbb{R}_{>0}$ is a delay such that, for all $\delta' < \delta$, no other rule can be applied to $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t + \delta', q, t_F)$,[6]

  where $c \overset{e/\text{op}(p)/e'}{\hookrightarrow_{\mathcal{E}_\varphi}} c'$ denotes the fact that the enforcement monitor moves from configuration c to configuration c' by reading e, executing operation op parameterised by p, and outputting e'.

[6] The allowed delays are obviously not known in the starting configuration of rule ***idle***. In practice, at the implementation level, allowed delays are determined using busy-waiting.

A configuration $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F)$ of the EM consists of the following elements: σ_{ms} is the sequence which is corrected and can be released as output; σ_{mc} is the input sequence read by the EM, but yet to be corrected, except for events that are suppressed; t indicates the current date; q is the current state of $[\![\mathcal{A}_\varphi]\!]$ reached after processing the sequence already released, followed by the timed word in memory σ_{ms}, i.e., $E_\varphi(\text{obs}(\sigma, t))$; t_F is the arrival date in q.

Semantic rules can be understood as follows:

- Upon the reception of an event (t, a) (i.e., when t is the date in the configuration and (t, a) is read), one of the following rules is executed. Notice that their conditions are exclusive of each others.
  - **1.** Rule ***store-***φ is executed if function update returns state $q' \in Q_F$, timed word w and marker `ok`, indicating that φ can be satisfied by the sequence already released as output, followed by σ_{ms}, and followed by w which minimally delays $\sigma_{\text{mc}} \cdot (t, a)$ to satisfy φ. When executing the rule, sequence w is appended to the content of output memory σ_{ms}, the input memory σ_{mc} is emptied, q' is the new state and $\text{end}(w)$ is the new arrival date.
  - **2.** Rule ***store***$_{\text{sup}}$***-***$\overline{\varphi}$ is executed if the update function returns marker `bad`, indicating that $\sigma_{\text{mc}} \cdot (t, a)$ followed by any sequence cannot be corrected. Event (t, a) is then suppressed, and the configuration remains unchanged.
  - **3.** Rule ***store-***$\overline{\varphi}$ is executed if the update function returns marker `c_bad`, indicating that $\sigma_{\text{mc}} \cdot (t, a)$ cannot be corrected yet. The event (t, a) is then appended to the internal memory σ_{mc}, but σ_{ms}, q and t_F remain unchanged.
- When no event can be received, one of the following rules is applied, with decreasing priority:
  - **4.** Rule ***release*** is executed if the current date t is equal to the date corresponding to the first event of the timed word $\sigma_{\text{ms}} = (t, a) \cdot \sigma'_{\text{ms}}$ in the memory. The event is released as output and removed from σ_{ms} in the resulting configuration.
  - **5.** Rule ***idle*** adds the time elapsed δ to the current value of t when neither store nor release operations are possible at any time instant between t and $t + \delta$.

Note, all rules except rule ***idle*** execute in zero time. Moreover, it is important to notice that the definition of update entails that the state q inside a configuration is either initial (initially $q = q_0$) or accepting (it is only modified by a ***store-***φ rule which makes it jump to a state $q \in Q_F$ as a result of update), one case not excluding the other (e.g., for safety properties).

Example 7 *(Execution of an enforcement monitor).* We illustrate how the rules of Definition 10 are applied to enforce property φ_3 (see Section 2), recognised by the automaton depicted in Fig. 6c with $\Sigma_3 = \{op_1, op_2, op\}$, and the input timed word $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. Fig. 11 shows how semantic rules are applied according to the current date t, and the evolution of the configurations of the enforcement monitor, together with input and output. More precisely, each line is of the form $O/c/I$, where O is the sequence of released events, c is a configuration, and I is the residual of the input σ after its observation at date t. The resulting (final) output is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$, which satisfies property φ_3. Note that after $t = 10$, only rule ***idle*** can be applied.

7.4. Relating enforcement functions and enforcement monitors

We show how the definitions of enforcement function and enforcement monitor are related: given a property φ, any input sequence σ, at any date t, the output of the associated enforcement function and the output behaviour of the associated enforcement monitor are equal.

Preliminaries We first describe how an enforcement monitor reacts to an input sequence. In the remainder of this section, we consider an enforcement monitor $\mathcal{E} = (C^{\mathcal{E}}, c_0^{\mathcal{E}}, \Gamma^{\mathcal{E}}, \hookrightarrow_{\mathcal{E}})$, not related to a property. Enforcement monitors, described in Section 7, are deterministic. By determinism, we mean that, given an input sequence, the observable output sequence is unique. Moreover, given $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, how an enforcement monitor reads σ until date t is unique: it goes through a unique sequence of configurations. Since rule ***idle*** does not read nor produce any event, ϵ belongs to the input alphabet. Thus, given an input sequence σ and a date t, there is possibly an infinite set of corresponding sequences over the *input-operation-output* alphabet (as in Definition 10). All these sequences are equivalent: they involve the same configurations for the enforcement monitor and the same output sequence. Consequently, the rules of transition relations are ordered in such a way that reading ϵ will always be the transition with least priority. Consequently, given an input sequence, reading ϵ (and doing other operations such as outputting some event) is always possible when the monitor cannot read an input.

More formally, let us define $\mathcal{E}^{\text{ioo}}(\sigma, t) \in (\Gamma^{\mathcal{E}})^*$ to be the unique sequence of transitions (triples comprised of an optional input event, an operation, and an optional output event) that is "triggered" from the initial configuration, when the enforcement monitor reads σ until date t:

Definition 11 *(Input-operation-output sequence).* Given an input sequence $\sigma \in \text{tw}(\Sigma)$ and some date $t \in \mathbb{R}_{\geq 0}$, we define the input-operation-output sequence, denoted as $\mathcal{E}^{\text{ioo}}(\sigma, t)$, as the unique[7] sequence of $(\Gamma^{\mathcal{E}})^*$ such that:

[7] The uniqueness of $\mathcal{E}^{\text{ioo}}(\sigma, t)$ is discussed in Remark 9 in Appendix A.3.

$$
\begin{array}{ll}
t=0 & \epsilon/(\epsilon,\epsilon,0,(l_0,0,0),0)/(2,op_1)\cdot(3,op_1)\cdot(3.5,op)\cdot(6,op_2)\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(2)/\epsilon\\
t=2 & \epsilon/(\epsilon,\epsilon,2,(l_0,0,0),0)/(2,op_1)\cdot(3,op_1)\cdot(3.5,op)\cdot(6,op_2)\\
 & \qquad\qquad \downarrow (2,op_1)/\text{store-}\overline{\varphi}(2,op_1)/\epsilon\\
t=2 & \epsilon/(\epsilon,(2,op_1),2,(l_0,0,0),0)/(3,op_1)\cdot(3.5,op)\cdot(6,op_2)\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(1)/\epsilon\\
t=3 & \epsilon/(\epsilon,(2,op_1),3,(l_0,0,0),0)/(3,op_1)\cdot(3.5,op)\cdot(6,op_2)\\
 & \qquad\qquad \downarrow (3,op_1)/\text{store}_{\text{sup}}\text{-}\overline{\varphi}(3,op_1)/\epsilon\\
t=3 & \epsilon/(\epsilon,(2,op_1),3,(l_0,0,0),0)/(3.5,op)\cdot(6,op_2)\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(0.5)/\epsilon\\
t=3.5 & \epsilon/(\epsilon,(2,op_1),3.5,(l_0,0,0),0)/(3.5,op)\cdot(6,op_2)\\
 & \qquad\qquad \downarrow (3.5,op)/\text{store-}\overline{\varphi}(3.5,op)/\epsilon\\
t=3.5 & \epsilon/(\epsilon,(2,op_1)\cdot(3.5,op),3.5,(l_0,0,0),0)/(6,op_2)\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(2.5)/\epsilon\\
t=6 & \epsilon/(\epsilon,(2,op_1)\cdot(3.5,op),6,(l_0,0,0),0)/(6,op_2)\\
 & \qquad\qquad \downarrow (6,op_2)/\text{store-}\varphi(6,op_2)/\epsilon\\
t=6 & \epsilon/((6,op_1)\cdot(8,op)\cdot(10,op_2),\epsilon,6,(l_0,4,2),10)/\epsilon\\
 & \qquad\qquad \downarrow \epsilon/\text{release}(6,op_1)/(6,op_1)\\
t=6 & (6,op_1)/((8,op)\cdot(10,op_2),\epsilon,6,(l_0,4,2),10)/\epsilon\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(2)/\epsilon\\
t=8 & (6,op_1)/((8,op)\cdot(10,op_2),\epsilon,8,(l_0,4,2),10)/\epsilon\\
 & \qquad\qquad \downarrow \epsilon/\text{release}(8,op)/(8,op)\\
t=8 & (6,op_1)\cdot(8,op)/((10,op_2),\epsilon,8,(l_0,4,2),10)/\epsilon\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(2)/\epsilon\\
t=10 & (6,op_1)\cdot(8,op)/((10,op_2),\epsilon,10,(l_0,4,2),10)/\epsilon\\
 & \qquad\qquad \downarrow \epsilon/\text{release}(10,op_2)/(10,op_2)\\
t=10 & (6,op_1)\cdot(8,op)\cdot(10,op_2)/(\epsilon,\epsilon,10,(l_0,4,2),10)/\epsilon\\
 & \qquad\qquad \downarrow \epsilon/\text{idle}(t)/\epsilon\\
t>10 & (6,op_1)\cdot(8,op)\cdot(10,op_2)/(\epsilon,\epsilon,t,(l_0,4,2),10)/\epsilon
\end{array}
$$

Fig. 11. Execution of an enforcement monitor for φ_3. The enforcement monitor ensures that if the automaton for φ_3 reads the (entire) output sequence, it remains in its accepting states.

$$
\begin{aligned}
&\exists c \in C^{\mathcal{E}} : c_0^{\mathcal{E}} \overset{\mathcal{E}^{\text{ioo}}(\sigma,t)}{\hookrightarrow}{}^{*}_{\mathcal{E}}\, c\\
&\quad \wedge\ \Pi_1(\mathcal{E}^{\text{ioo}}(\sigma,t)) = \text{obs}(\sigma,t)\\
&\quad \wedge\ \text{timeop}(\Pi_2(\mathcal{E}^{\text{ioo}}(\sigma,t))) = t\\
&\quad \wedge\ \neg\left(\exists c' \in C^{\mathcal{E}}, e \in (\mathbb{R}_{\geq 0}\times\Sigma) : c \overset{(\epsilon,\text{release}(e),e)}{\hookrightarrow}_{\mathcal{E}} c'\right),
\end{aligned}
$$

where the timeop function indicates the duration of a sequence of enforcement operations and says that only the idle enforcement operation consumes time. Formally:

$$
\begin{aligned}
\text{timeop}(\epsilon) &= 0;\\
\text{timeop}(op\cdot ops) &= \begin{cases} d + \text{timeop}(ops) & \text{if } \exists d \in \mathbb{R}_{>0} : op = \text{idle}(d),\\ \text{timeop}(ops) & \text{otherwise.}\end{cases}
\end{aligned}
$$

The observation of the input timed word σ at any date t, corresponding to $\text{obs}(\sigma,t)$, is the concatenation of all the input events read/consumed by the enforcement monitor over various steps. Observe that, because of the assumptions on $\Gamma^{\mathcal{E}}$, only

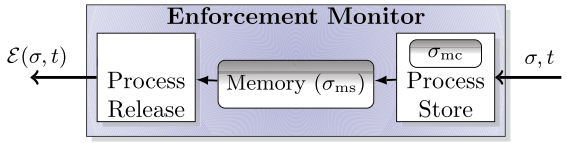


Fig. 12. Realising an EM.

rule ***idle*** applies to configuration c: rule ***release*** does not apply by definition of $\mathcal{E}^{\text{ioo}}(\sigma, t)$ and none of the ***store*** rules applies because $\Pi_1(\mathcal{E}^{\text{ioo}}(\sigma, t)) = \text{obs}(\sigma, t)$.

Relating enforcement functions and enforcement monitors We now relate the enforcement function E_φ and the enforcement monitor $\mathcal{E}_\varphi$, for a property φ, using the input-operation-output behaviour $\mathcal{E}^{\text{ioo}}_\varphi$ of $\mathcal{E}_\varphi$ as per Definition 11. Seen from the outside, an enforcement monitor $\mathcal{E}_\varphi$ behaves as a device reading and producing timed words. Overloading notations, this input/output behaviour can be characterised as a function $\mathcal{E}_\varphi : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to \text{tw}(\Sigma)$ defined as:

$$\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathcal{E}_\varphi(\sigma, t) = \Pi_3\left(\mathcal{E}^{\text{ioo}}_\varphi(\sigma, t)\right).$$

The corresponding output timed word $\mathcal{E}_\varphi(\sigma, t)$, at any date t, is the concatenation of all the output events produced by the enforcement monitor over various steps of the enforcement monitor (where all ϵ's are erased through concatenation). In the following, we do not distinguish between an enforcement monitor and the function that characterises its behaviour.

Finally, we define an implementation relation between enforcement monitors and enforcement functions as follows.

Definition 12 *(Implementation relation).* Given an enforcement function E_φ (as per Definition 8) and an enforcement monitor (as per Definition 10) whose behaviour is characterised by a function $\mathcal{E}_\varphi$, we say that $\mathcal{E}_\varphi$ *implements* E_φ iff:

$$\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \text{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\sigma, t).$$

Proposition 3 *(Relation between enforcement function and enforcement monitor). Given a property φ, its enforcement function E_φ (as per Definition 8, p. 14), and its enforcement monitor $\mathcal{E}_\varphi$ (as per Definition 10, p. 21), $\mathcal{E}_\varphi$ implements E_φ in the sense of Definition 12.*

Proof of Proposition 3 – sketch only. The proof is given in Appendix A.4, p. 37. The proof relies on an induction on the length of the input word σ. The induction step uses a case analysis, depending on whether the input is completely observed or not at date t, whether the input can be delayed into a correct output or not, and whether the memory content (σ_{ms}) is completely released or not at date t. The proof also uses several intermediate lemmas that characterise some special configurations (e.g., value of the clock variable, content of the memory σ_{ms}) of an enforcement monitor. □

8. Enforcement algorithms: implementation of enforcement mechanisms

An enforcement monitor remains an abstract view of a real enforcement mechanism, and needs to be further concretised into an implementation. The implementation of an enforcement monitor consists of two processes running concurrently (called hereafter StoreProcess and ReleaseProcess) and started simultaneously, and a shared memory, as shown in Fig. 12. StoreProcess implements the ***store*** rules of the enforcement monitor. The memory contains the timed word σ_{ms}: the corrected sequence that can be released as output. The memory is realised as a queue, shared between the StoreProcess and ReleaseProcess, where the StoreProcess adds events, which are processed and corrected, to this queue. ReleaseProcess reads the events stored in the memory σ_{ms} and releases the action corresponding to each event as output, when time reaches the date associated to the event. StoreProcess also makes use of another internal buffer σ_{mc} (not shared with any other process), to store the events which are read, but cannot be corrected yet, to satisfy the property. In the algorithms, primitive await is used to wait for a trigger event from another process or to wait until some condition becomes true. Primitive wait is used by a process to wait for some amount of time determined by the process itself.

In the following, we first present algorithm update used by algorithm StoreProcess, then present algorithm StoreProcess, and finally algorithm ReleaseProcess.

Algorithm update *(see Algorithm 1)* Algorithm update implements function update from Definition 9. It takes as input q, the current state, t_F, the arrival date in state q, the events stored in the internal memory σ_{mc} of StoreProcess, and the new event (t, a), and returns a new state q', a timed word σ'_{mc}, and a marker in the set $\{\texttt{ok}, \texttt{c\_bad}, \texttt{bad}\}$, indicating whether $\sigma_{\text{mc}} \cdot (t, a)$ can be delayed to satisfy φ.

The algorithm makes use of the following functions. Function computeReach computes all the reachable paths[8] from the current state q upon events in $\sigma_{\text{mc}} \cdot (t, a)$ that start after date t, where time starts at date t_F, the arrival date

[8] A path is a run in the symbolic (zone) graph.

Algorithm 1 update$(q, t_F, \sigma_{\text{mc}}, (t, a))$.

```
allPaths ← computeReach(σ_mc · (t, a), q, t_F)
accPaths ← getAccPaths(allPaths)
if accpaths ≠ ∅ then
   σ'_mc ← getOptimalWord(accPaths, σ_mc · (t, a))
   return(post(q, σ'_mc), σ'_mc, ok)
else
   isReachable ← checkReachAcc(allPaths)
   if isReachable = ff then
      return(q, σ_mc, bad)
   else
      return(q, σ_mc, c_bad)
   end if
end if
```

Algorithm 2 StoreProcess.

```
t ← 0
(q, t_F) ← (q_0, 0)
(σ_ms, σ_mc) ← (ϵ, ϵ)
while tt do
   (t, a) ← await(event)    /* i.e., action a is received at date t */
   (q', σ'_mc, isPath) ← update(q, t_F, σ_mc, (t, a))
   if isPath = ok then
      σ_ms ← σ_ms · σ'_mc
      σ_mc ← ϵ
      q ← q'
      t_F ← end(σ'_mc)
   else
      σ_mc ← σ'_mc
   end if
end while
```

in q. Formally, it computes $\left\{w \in \text{tw}(\Sigma) \mid q \xrightarrow{w}_{t_F}\right\} \cap \text{CanD}(\sigma_{\text{mc}} \cdot (t, a))$. Function getAccPaths takes as input all the paths returned by computeReach and returns only those that lead to a state in Q_F. Formally it computes $k_{Q_F}(q, t_F, \sigma_{\text{mc}} \cdot (t, a)) = \left\{w \in \text{tw}(\Sigma) \mid q \xrightarrow{w}_{t_F} Q_F\right\} \cap \text{CanD}(\sigma_{\text{mc}} \cdot (t, a))$. Both functions use forward analysis, zone abstraction, and operations on zones such as the resetting of clocks and intersection of guards [16].

Function getOptimalWord takes all the accepting paths and a sequence $\sigma_{\text{mc}} \cdot (t, a)$ and computes optimal delays for events in $\sigma_{\text{mc}} \cdot (t, a)$. This function first computes an optimal date for each event, for all accepting paths. Finally, it picks a path among the set of accepting paths whose ending date is minimal, and returns it as the result. Function getOptimalWord implements the computation described in Section 7.2 (§ *On the computation of function* update) using a simplified version of the algorithm in [19]. Function post takes a state of the automaton defining the property, a timed word, and computes the state reached by the automaton. Function checkReachAcc takes a set of paths as input. From the last state in each path, it checks if an accepting state in the input TA is reachable or not (i.e., whether a state in Q_F is reachable). It returns `tt`, if an accepting state is reachable, and `ff` otherwise. Formally it checks whether $k_{Q_F \cup B^c}(q, t_F, \sigma)$ is empty.

The algorithm proceeds as follows. If the set of accepting paths is not empty (i.e., a state in Q_F is reachable upon delaying $\sigma_{\text{mc}} \cdot (t, a)$), then function update returns `ok`, the optimal word computed using getOptimalWord, and the state reached in the TA (computed using the function post). Otherwise, it checks if it is possible to reach an accepting state in the future (computed using function checkReachAcc). If it is impossible to reach an accepting state (i.e., from all the states reached upon delaying $\sigma_{\text{mc}} \cdot (t, a)$, Q_F is not reachable), then function update returns `bad`, σ_{mc}, and the current state q. Otherwise, it returns the current state q, $\sigma_{\text{mc}} \cdot (t, a)$, and `c_bad`.

Algorithm StoreProcess (see Algorithm 2) Algorithm StoreProcess is an infinite loop that scrutinises the system for input events. In the algorithm, q represents the state of the property automaton.

The algorithm proceeds as follows. StoreProcess initially sets its clock t to 0. This clock keeps track of the time elapsed and increases with physical time. Variable t_F is initialised to 0. This variable contains the date of the last event of σ_{ms}, if σ_{ms} is not empty, and the date of the last released event otherwise. The algorithm also initialises q to q_0, and the two memories σ_{ms} and σ_{mc} to ϵ. It then enters an infinite loop where it waits for an input event (await(*event*)). When receiving an action a at date t, it stores event (t, a). It then invokes function update with the current state q, the arrival date t_F, the events stored in σ_{mc} and the new event (t, a). Then, function update returns a new state q', a timed word σ'_{mc} and the marker *isPath*. If marker $\textit{isPath} = \texttt{ok}$, it means that $\sigma_{\text{mc}} \cdot (t, a)$ can be corrected into the timed word σ'_{mc} computed by update and this word leads from state q to state q' in the underlying semantics of the timed automaton, at date $\text{end}(\sigma'_{\text{mc}})$. Then, timed word σ'_{mc} is appended to shared memory σ_{ms} (since σ'_{mc} leads to an accepting state q' from state q), the internal memory σ_{mc} is cleared, state q is updated to q' and t_F to $\text{end}(\sigma'_{\text{mc}})$. In all other cases, σ_{mc} is set to σ'_{mc}, the result of update, which is either σ_{mc} if $\textit{isPath} = \texttt{bad}$ (it is impossible to correct the input sequence σ_{mc} whatever are the future

Algorithm 3 ReleaseProcess.

```
d ← 0
while tt do
  await(σ_ms ≠ ϵ)
  (t, a) ← dequeue(σ_ms)
  wait(t − d)
  release(a)
end while
```

events) or $\sigma_{\text{mc}} \cdot (t, a)$ if $isPath = \texttt{c\_bad}$. Event (t, a) is thus deleted. In both cases, state q, t_F and memory σ_{ms} are not modified.

Algorithm ReleaseProcess (see Algorithm 3) Algorithm ReleaseProcess is an infinite loop that scrutinises memory σ_{ms} and releases actions as output.

The algorithm proceeds as follows. Initially, clock d, which keeps track of the time elapsed, is set to 0 and then increases with physical time. ReleaseProcess waits until the memory is not empty ($\sigma_{\text{ms}} \neq \epsilon$). Using operation dequeue, the first element stored in the memory is removed, and is stored as (t, a). Since d time units elapsed, process ReleaseProcess waits for $(t - d)$ time units before performing operation release(a), releasing action a as output at date t (which amounts to appending (t, a) to the output of the enforcement monitor).

Remark 8 *(Launching StoreProcess and ReleaseProcess).* In order to respect the semantics of the enforcement monitor, the two processes StoreProcess and ReleaseProcess should be launched simultaneously. This ensures that their current dates (encoded by t for StoreProcess and d for ReleaseProcess) are always equal.

9. Implementation and evaluation

We implemented the algorithms in Section 8 and developed an experimentation framework called TiPEX: (Timed Properties Enforcement during eXecution)[9] in order to:

1. validate through experiments the architecture and feasibility of enforcement monitoring, and
2. measure and analyse the performance of the update function of the StoreProcess.

From [5], we completely re-implemented the synthesis of enforcement monitors. TiPEX supports now all regular properties. The prototype presented in [5] handles only safety and co-safety properties, with independent algorithms and prototype implementations for each class. Now, following the algorithms proposed in this paper, TiPEX supports all regular properties defined by deterministic one-clock timed automata. In [20], we describe the implementation of a simplified version of function update that does not allow to suppress events. We recently implemented another version of function update based on the enforcement mechanisms and algorithms described in Section 8. In this section, we compare the performance of the implementations of these functions. Note, when we consider suppression, when an accepting state is not reachable with the events received so far, we need to perform another additional computationally-expensive analysis (checkReachAcc in Algorithm 1), to decide whether or not the last event should be suppressed.

The rest of this section is organised as follows. Section 9.1 describes our experimental framework. Section 9.2 present the properties used in the evaluation. Section 9.3 discusses the evaluation results.

9.1. Experimental framework

The experimental framework is depicted in Fig. 13. As mentioned in [5], regarding algorithm StoreProcess, the most computationally intensive step is the call to function update. We thus focus on this function in the evaluation.

Module Main uses module Trace Generator that provides a set of input traces to test the module Store. Module Trace Generator takes as input the alphabet of actions, the range of possible delays between actions, the desired number of traces, and the increment in length per trace. For example, if the number of traces is 5 and the increment in length per trace is 100, then 5 traces will be generated, where the first trace is of length 100 and the second trace of length 200 and so on. For each event, Trace Generator picks an action (from the set of possible actions), and a random delay (from the set of possible delays) which is the time elapsed after the previous event or the system initialisation for the first event. For this purpose, Trace Generator uses methods from the Python random module.

Module Store takes as input a property and one trace, and returns the total execution time of the update function to process the given input trace. The TA modelling the property is a UPPAAL [21] model written in XML. Module Store uses the `pyuppaal` library to parse the UPPAAL model (input property), and the UPPAAL DBM library to implement the update

[9] Available at http://srinivaspinisetty.github.io/Timed-Enforcement-Tools/.

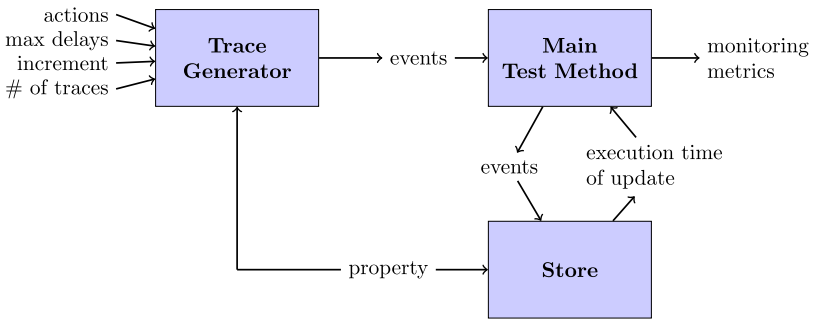


Fig. 13. Experimental framework.

Table 1
Performance analysis of enforcement monitors for φ_s.

$\|tr\|$	$t_$update	$t_$update-sup	*mem*	*mem*-sup
10,000	6.44	6.64	17.8	17.9
20,000	12.73	13.44	19.6	19.6
30,000	19.51	20.16	21.3	21.3
40,000	26.41	26.50	22.6	22.7
50,000	31.88	33.10	24.3	24.3
60,000	38.44	39.84	26.2	26.2
70,000	45.16	45.92	27.7	27.8
80,000	51.21	53.34	29.1	29.1

function.[10] The sequence of events received by the enforcement monitor is modelled by a second UPPAAL model. Module Main Test Method sends this sequence to module Store (using the property), and keeps track of the result returned by the Store module for each trace.

Experiments were conducted on an Intel Core i5-4210U at 1.70GHz CPU, with 4 GB RAM, and running on Ubuntu 14.04 LTS.

9.2. Description of the properties

We describe the properties used in our experiments and discuss the results of the performance analysis.

The properties follow different patterns [22], and belong to different classes. They are inspired from the properties introduced in Example 1. They are recognised by one-clock timed automata since this is a limitation of our current implementation (extension to more than one clock is ongoing). We however expect the trends exposed in the following to be similar when the complexity of automata grows, since it induces heavier computation for each call to function update.

- Property φ_s is a safety property expressing that "*There should be a delay of at least* 5 *time units (t.u.) between any two request actions*".
- Property φ_{cs} is a co-safety property expressing that "*A request should be immediately followed by a grant, and there should be a delay of at least* 6 *t.u. between them*".
- Property φ_{re} is a regular property, but neither a safety nor a co-safety property, and expresses that "*Resource grant and release should alternate. After a grant, a request should occur between 15 to 20 t.u.*".

The automata defining the above properties can be found in [23].

9.3. Performance evaluation of function update

Results of the performance analysis for the properties are reported in Tables 1, 2, and 3. The reported numbers are mean values over 10 runs. Note, 10 runs were sufficient to obtain 95% confidence for all metrics, and the measurement error was less than 1%. The entry $|tr|$ indicates the length of the input trace (i.e., the number of events input to the enforcement monitor). The entry $t_$update (resp. $t_$update-sup) indicates the total execution time of the function update without (resp. with) suppression in seconds. The entry *mem* (resp. *mem*-sup) indicates the maximum memory used by the Main Test Method when using function update without (resp. with) suppression; both measured in megabytes.

[10] The `pyuppaal` and `DBM` libraries are provided by Aalborg University and can be downloaded at http://people.cs.aau.dk/~adavid/python/.

Table 2
Performance analysis of enforcement monitors for φ_{re}.

$\lvert tr\rvert$	$t_$update	$t_$update-sup	*mem*	*mem*-sup
10,000	10.21	20.33	17.6	17.6
20,000	20.56	39.32	19.0	19.0
30,000	30.95	61.20	20.2	20.2
40,000	42.37	82.23	21.6	21.6
50,000	53.67	101.46	22.8	22.8
60,000	62.06	121.55	24.2	24.2
70,000	81.63	137.49	25.4	25.4
80,000	91.89	167.16	26.8	26.8

Table 3
Performance analysis of enforcement monitors for φ_{cs}.

$\lvert tr\rvert$	$t_$update	$t_$update-sup	*mem*	*mem*-sup
100	2.022	2.256	16.4	16.4
200	8.124	8.547	16.4	16.4
300	18.207	18.868	16.4	16.4

Strategy for generating traces To have a meaningful performance assessment of function update, module Trace Generator uses a strategy to ensure that calls to function update yields computation using σ_{mc}. For (the safety) property φ_{s}, module Trace Generator generates events so that each event of the trace leads to a call to function update to correct the date of the input event. This strategy allows to assess the performance of function update when it is extensively used with buffer σ_{mc} empty. For (the co-safety) property φ_{cs}, module Trace Generator ensures that input sequences can be corrected only on the last event (hence implying that, for a sequence of length n, function update is called n times where the buffer containing σ_{mc} is of size $i-1$ on the i^{th} call). This strategy allows to assess the performance of function update when σ_{mc} is used significantly. For (the regular property) φ_{re}, module Trace Generator ensures that the property can be corrected every two events, which is the length of the minimal path between accepting locations of the underlying automaton of φ_{re}. This strategy allows to asses the performance of function update when alternating between finding a correction of the input sequence using buffer σ_{mc} and buffering corrected events in buffer σ_{ms}.

Safety property φ_{s} (see Table 1) We can observe that $t_$update, and $t_$update-sup increase linearly with the length of the input trace. Moreover, the time taken per call to update (i.e., $t_$update$/\lvert tr\rvert$) does not depend on the length of the trace. This behaviour is as expected for a safety property. Indeed, function update is always called with only one event which is read as input (the internal buffer σ_{mc} remains empty). Consequently, the state of the TA is updated after each event, and after receiving a new event, the possible transitions leading to a good state from the current state are explored. For the same input trace, there is no significant variation in the values of $t_$update, and $t_$update-sup. This behaviour is as expected because for the considered safety property (φ_{s}) and input traces, after receiving a new event, it is always possible to compute a delay to satisfy the property. Thus, in the function update with suppression, checkReachAcc is never invoked.

Regarding memory usage, we can notice that by increasing the length of the input trace by 10,000, the peak memory usage increases by less than 2 MB. For the same input trace, there is no variation in memory usage (*mem* and *mem*-sup are equal).

Regular property φ_{re} (see Table 2) Recall that the considered input traces are generated in such a way that they can be corrected every two events. Consequently, function update is invoked with either one or two events. For the considered input traces, the time taken per call to function update does not depend on the length of the trace. Moreover, for input traces of same length, the value of t_update (resp. $t_$update-sup) is higher for φ_{re} than the value of t_update (resp. $t_$update-sup) for φ_{s}. This stems from the fact that, for a safety property, function update is invoked only with one event. Furthermore, for the same input trace, $t_$update-sup is greater than $t_$update. This stems from the fact that, for the considered input traces (where it is possible to correct every two events) the function update with suppression invokes function checkReachAcc $\lvert tr\rvert/2$ times.

Regarding memory usage, by increasing the length of the input trace by 10,000, the peak memory usage increases by less than 2 MB. For input traces of same length, there is no significant variation in the values of *mem* and *mem*-sup between φ_{re} and φ_{s}.

Co-safety property φ_{cs} (see Table 3) Recall that the considered input traces are generated in such a way that they can be corrected only upon the last event. From the results presented in Table 3, notice that $t_$update, and $t_$update-sup are now quadratic. Moreover, the average time per call to function update increases with $\lvert tr\rvert$. For the considered input traces, this behaviour is as expected for a co-safety property because the length of the internal buffer σ_{mc} increases after each event, and thus function update is invoked with a growing sequence.

For the same input trace, $t_$update-sup is greater than $t_$update. For example, for input traces of length 100, $t_$update-sup is around 0.2 seconds greater than $t_$update. Indeed, for the considered input traces (where it is possible to correct the input sequence only upon the last event) the function update with suppression invokes function checkReachAcc $|tr| - 1$ times. We can also observe that $t_$update-sup $-t_$update increases linearly with $|tr|$.

Regarding memory usage, since we consider small increments of the input traces, we cannot notice significant variation. For input trace of length 100, peak memory usage noticed for φ_s is 16.5 MB. Thus we can notice that, for input traces of same length, for φ_{re}, φ_s, and φ_{cs}, there is no significant variation in the value of *mem*.

10. Related work

Several approaches for the runtime verification and enforcement of properties are related to the one proposed in this paper.

10.1. Runtime verification

As a verification/validation technique, runtime enforcement is related to runtime verification. At an abstract level, a runtime verification approach consists in synthesising a verification monitor (cf. [24]), i.e., a decision procedure used at runtime. The monitor observes the system under scrutiny and emits verdicts regarding the satisfaction or violation of the property of interest. See [25–28] for short tutorials and surveys on runtime verification. Runtime verification principles have been used in many concrete application domains and for various purposes such as the safety checking of cyber-physical systems [29–32], the security of financial and IT systems [33–35], and many more.

10.2. Runtime verification of timed properties

We discuss more specifically some approaches for the runtime verification of timed properties for real-time systems. One can also refer to the survey of Goodloe and Pike [36] which presents some approaches to monitoring hard real-time systems and potential application-domains when monitoring safety properties.

Several approaches consider the problem of synthesising automata-based monitors from formulae in temporal logics that handle physical time (as opposed to logical time). Sokolsky et al. [37] introduced an expressive first-order logic tailored for runtime verification. The logic features event attributes (aka parametric events) and dynamic indexing of properties (to handle the dynamic creation of monitors at runtime). Models of the logic also refers to physical time. Bauer et al. [38] synthesised monitors for timed-bounded properties expressed in a variant of Timed Linear Temporal Logic tailored for monitoring. Nickovic et al. [17,18] synthesised timed automata from Metric Temporal Logic (a temporal logic with a dense notion of time). Still for MTL, Thati [39] use rewriting of formulae for online monitoring. All these approaches are compatible with ours since they are purposed to synthesise decision procedures for logic-based timed specification formalisms. More specifically, the synthesised automata-based monitors can be used as input in our approach as replacements of timed automata.

Basin et al [40] provided a general comparison of monitoring algorithms for real-time systems. Time models are categorised as i) either point-based algorithms or interval-based, and ii) either dense or discrete depending on the underlying ordering of time points (i.e., finitely or infinitely many time points). Basin et al. presented and compared monitoring algorithms for the past-only fragment of propositional metric temporal logic.

Several tools have been proposed for monitoring timed properties. RT-MaC [41] verifies timeliness and reliability correctness properties at runtime. The Analog Monitoring Tool [42] verifies formulae in Signal Temporal Logic over continuous signals. LARVA [43,44] verifies properties (over Java programs) expressed in several specification formalisms by translating input specifications into the so-called Dynamic Automata with Timers and Events which basically resemble timed automata with stop watches. Contrary to these approaches, the monitors presented in this paper differ in their objectives and how they are interfaced with the system: the monitors are not intended to modify the internal state of the system but rather to modify a sequence of timed events between two systems.

10.3. Runtime enforcement of untimed properties

Roughly speaking, the research efforts in runtime enforcement aims at defining and implementing enforcement primitives that supplement the monitors used in runtime verification. Most of the work in runtime enforcement was dedicated to untimed properties (see [45] for a short overview). Schneider introduced security automata as the first runtime mechanism for enforcing safety properties [1]. Ligatti et al. [3] later introduced edit-automata as enforcement monitors. Edit-automata can insert a new action by replacing the current input, or suppress it. Similar to edit-automata are generic enforcement monitors [4] which are finite-state machines augmented with a memory and parameterised with enforcement primitives operating on the input and memory. Moreover, some variants of edit-automata differ in how they ensure the transparency constraints (see e.g., [46]). Synthesis techniques of enforcement mechanisms from a property have been proposed only for generic enforcement monitors [4] and restricted forms of edit-automata [3].

Note, several runtime verification tools allow the user to define some treatment of errors through the (manual) definition of some form of enforcement primitives. For instance, JavaMOP and the RV system [47] define the notion of *code handler* which are user-defined code-snippets that can be attached to monitor states. LARVA allows the user to specify corrective actions [48] that can be used for undoing the effects of previous actions carried out by the system.

10.4. Runtime enforcement of timed properties

The endeavours on runtime enforcement discussed in the previous subsection consider logical time, as opposed to physical time. Moreover, storing an event is assumed without consequence on the execution nor on the satisfiability of the property, i.e., the duration during which an event is retained in memory has no influence. In the following of this subsection, we discuss the approaches on runtime enforcement that consider physical time.

Basin et al. [49] refined the work of Schneider on security automata to take into account discrete-time constraints by modelling the passing of time as uncontrollable events. Similarly, we consider elapsing of time as uncontrollable but consider dense time. The enforcement mechanisms in [49] differ from ours in several aspects: they consider only truncation automata (and they are thus limited to safety properties, not necessarily regular). Moreover, our enforcement mechanisms have additional enforcement primitives: buffering of actions (which basically amounts to letting time elapse) and suppression of actions which allows for longer inputs to be processed by enforcement mechanisms.

In previous work [5,9], we introduced the problem of runtime enforcement for timed properties. We similarly proposed several notions of enforcement mechanisms: enforcement function, enforcement monitor, and enforcement algorithms. In [5], only safety and co-safety properties are considered and different definitions of mechanisms are proposed for each class. In [9], all regular properties are considered. Given a timed automaton, enforcement functions, monitors and algorithms are synthesised according to one general definition. Also, for the enforcement of co-safety properties, the approach in [5] assumes that time elapses differently for input and output sequences (the sequences are de-synchronised). More precisely, the delay of the first event of the output sequence is computed from the moment an enforcement mechanism detects that its input sequence can be corrected (that is, the mechanism has read a sequence that can be delayed into a correct sequence). Compared to [5], the approaches in [9] and this paper are more realistic as they do not suffer from this "shift" problem.

10.5. Monitorability and enforceability

In this paper, we identify some timed properties that are not enforceable by mechanisms that comply to the constraints mentioned in Section 5.2 (see Example 6). Characterising monitorable properties (i.e., properties that can be runtime verified) and enforceable properties are two important endeavours. We briefly discuss some of the main approaches on these topics in the following and discuss in Section 11.2 how we plan to characterise enforceable timed properties in the future.

Monitorable properties Kim et al. [50] first defined monitorable properties as the co-recursively enumerable safety properties. Pnueli et al. [51] generalised the definition to the properties for which it is always possible to determine a definitive satisfaction or violation at runtime. Bauer et al. [38] showed that safety and co-safety properties are monitorable in the sense of [51]. Later, Falcone et al. [10] showed that obligation properties form a strict subset of the set of monitorable properties in the sense of [51], but that less properties should be monitored in practice. Sistla et al. [52] defined necessary and sufficient conditions for the monitorability of hybrid systems where an Extended Hidden Markov system is monitorable if there exists an arbitrarily-precise monitor stating verdicts on the system outputs. More recently, Rosu [53] defined monitorable properties as safety properties arguing that these can be specified by general (finite-state machine) monitors.

Enforceable properties Enforceable properties are the properties for which a sound and transparent enforcement monitor can be synthesised. The set of enforceable properties depends on the primitives conferred to enforcement monitors. Security automata [1] can enforce safety properties. Note, Schneider, Hamlen, and Morrisett [2] showed that security automata can only monitor co-recursively enumerable safety properties because of computational limits exhibited by Viswanathan and Kim [54]. Edit-automata [3] can enforce infinite renewal properties. (The set of infinite renewal properties is a superset of safety properties and contains some liveness properties.) Generalised enforcement monitors [4] can enforce response properties in the safety-progress classification. In addition to enforcement primitives and computability constraints, enforceability limitations arise when properties are expressed over infinite sequences (see [45,4] for a comparison of enforceable untimed properties over infinite sequences). However, any property over finite sequences is enforceable with a monitor endowed with the primitives of an edit-automaton (see Section 10.3 and [3]) [10]. More recently, Basin et al. [49] showed that security automata can enforce the safety properties that cannot be violated through a sequence of uncontrollable events.

11. Conclusions

11.1. Summary

This paper presents a general enforcement monitoring framework for systems with timing requirements. We show how to synthesise enforcement mechanisms for any regular timed property (modelled by a timed automaton). The enforcement

mechanisms proposed in this paper are more powerful than the ones in our previous research endeavours [5,9]. In particular, in this paper, we propose enforcement mechanisms that delay the absolute dates of events of the observed input (while being allowed to shorten the delay between some events). Moreover, suppressing events is also introduced. An event is suppressed if it is not possible to satisfy the property by delaying, whatever are the future continuations of the input sequence (i.e., the underlying TA can only reach non-accepting states from which no accepting state can be reached). Formalising suppression required us to revisit the formalisation of all enforcement mechanisms. Enforcement mechanisms are described at several levels of abstraction (enforcement function, monitor, and algorithms), thus facilitating the design and implementation of such mechanisms. We propose a prototype implementation and our experiments demonstrate the feasibility of enforcement monitoring for timed properties.

11.2. Future work

Several avenues for future work are open by this paper.

First, we believe it is important to study and delineate the set of *enforceable timed properties*. As shown informally by this paper, some timed properties should be characterised as non-enforceable. For this purpose, an enforceability condition should be defined and used to delineate enforceable properties. Such a criterion should also ideally be expressible on timed automata. Note however that, even for non-enforceable properties, enforcement monitors can be built, but may not be able to output some correct input sequences. The output sequences of our enforcement mechanisms are however always either correct or empty.

Specifications are currently modelled with timed automata. One can consider synthesising enforcement mechanisms from more expressive formalisms. For instance, we could consider formalisms such as context-free timed languages (which can be useful for recursive specifications) or introduce data into requirements (which can be useful in some application domains, as shown for safety properties in [8]).

Implementing efficient enforcement monitors is another important aspect and should be done in a particular application domain. We propose TiPEX, a Python implementation of enforcement mechanisms with the objectives of i) making a quick prototype that shows feasibility of enforcement monitoring in a timed context, and ii) reusing some existing UPPAAL libraries. In the future, we will consider implementing our enforcement monitors in other languages such as C or Java, and we expect even better performance and a more stand-alone implementation.

Acknowledgements

The authors would like to thank the anonymous reviewers for their remarks and suggestions on an early version of this article.

The work reported in this article has been done in the context of the COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

Appendix A. Proofs

Recall that $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ is defined as:

$$E_\varphi(\sigma) = \Pi_1\big(\mathrm{store}_\varphi(\sigma)\big),$$

where $\mathrm{store}_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma) \times \mathrm{tw}(\Sigma)$ is defined as

$$\begin{aligned}
\mathrm{store}_\varphi(\epsilon) &= (\epsilon, \epsilon)\\
\mathrm{store}_\varphi(\sigma \cdot (t,a)) &= \begin{cases} (\sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}} \kappa_\varphi(\sigma_s, \sigma_c'), \epsilon), & \text{if } \kappa_\varphi(\sigma_s, \sigma_c') \neq \emptyset,\\ (\sigma_s, \sigma_c) & \text{if } \kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c') = \emptyset\\ (\sigma_s, \sigma_c') & \text{otherwise,} \end{cases}\\
&\quad \text{with } \sigma \in \mathrm{tw}(\Sigma), t \in \mathbb{R}_{\geq 0}, a \in \Sigma,\\
&\quad (\sigma_s, \sigma_c) = \mathrm{store}_\varphi(\sigma), \text{ and } \sigma_c' = \sigma_c \cdot (t,a)
\end{aligned}$$

where:

$$\kappa_\varphi(\sigma_s, \sigma_c') \stackrel{\mathrm{def}}{=} \mathrm{CanD}(\sigma_c') \cap \sigma_s^{-1} \cdot \varphi,$$

as defined in Section 6.2, with:

$$\mathrm{CanD}(\sigma) = \{w \in \mathrm{tw}(\Sigma) \mid w \succcurlyeq_d \sigma \wedge \mathrm{start}(w) \geq \mathrm{end}(\sigma)\},$$

as defined in Section 6.1.

A.1. Proof of Proposition 1 (p. 15)

We shall prove that, given a property $\varphi \subseteq \mathrm{tw}(\Sigma)$, the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, defined as per Definition 8 (p. 14), satisfies the physical constraint, is sound and transparent. These constraints are recalled below:

- **Physical constraint:**

$$\forall \sigma, \sigma' \in \mathrm{tw}(\Sigma) : \sigma \preccurlyeq \sigma' \implies E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma') \qquad (\textbf{Phy}).$$

- **Soundness:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon \qquad (\textbf{Snd}).$$

- **Transparency:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \lhd_d \sigma \qquad (\textbf{Tr}).$$

The proof of **(Phy)** is straightforward by noticing that function store_φ is monotonic on its first output ($\forall \sigma, \sigma' \in \mathrm{tw}(\Sigma) : \sigma \preccurlyeq \sigma' \implies \Pi_1(\mathrm{store}_\varphi(\sigma)) \preccurlyeq \Pi_1(\mathrm{store}_\varphi(\sigma'))$).

We now prove both **(Snd)** and **(Tr)** by an induction on the length of the input timed word σ. For this purpose, we actually prove a slightly stronger property of E_φ: for any $\sigma \in \mathrm{tw}(\Sigma)$, (i) E_φ satisfies $(\textbf{Snd})_\sigma \stackrel{\text{def}}{=} E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon$ and $(\textbf{Tr})_\sigma \stackrel{\text{def}}{=} E_\varphi(\sigma) \lhd_d \sigma$, and (ii) $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$, where σ_s and σ_c are as in the definition of $\mathrm{store}_\varphi()$, recalled above.

Induction basis ($\sigma = \epsilon$) The proof of the induction basis is immediate from the definitions of E_φ, $\mathrm{store}_\varphi(\epsilon)$, $\lhd$, and $\lhd_d$.

Induction step Let us suppose that for some $\sigma \in \mathrm{tw}(\Sigma)$, $E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon$ $(\textbf{Snd})_\sigma$, and $E_\varphi(\sigma) \lhd_d \sigma$ $(\textbf{Tr})_\sigma$ (induction hypothesis). Let us consider $\sigma' = \sigma \cdot (t, a)$, with $t \in \mathbb{R}_{\geq 0}$, $t \geq \mathrm{end}(\sigma)$, and $a \in \Sigma$. Suppose that $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma_c' = \sigma_c \cdot (t, a)$, where $\mathrm{end}(\sigma_c) \leq t$. We distinguish two cases:

- Case $\kappa_\varphi(\sigma_s, \sigma_c') \neq \emptyset$. In this case, we have $E_\varphi(\sigma \cdot (t,a)) = \Pi_1\big(\mathrm{store}_\varphi(\sigma \cdot (t,a))\big) = \sigma_s \cdot \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} \kappa_\varphi(\sigma_s, \sigma_c')$. From the definition of function κ_φ, we have $\kappa_\varphi(\sigma_s, \sigma_c') \subseteq \sigma_s^{-1} \cdot \varphi$, and thus $E_\varphi(\sigma \cdot (t,a)) \in \varphi$. Thus E_φ satisfies $(\textbf{Snd})_{\sigma'}$.
  From the induction hypothesis, we know that $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$. We deduce $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$ which shows that (ii) holds again for σ'.
  Let $w \in \kappa_\varphi(\sigma_s, \sigma_c')$. From the definition of $\kappa_\varphi()$, since $w \in \sigma_s^{-1} \cdot \varphi$, we have $\mathrm{start}(w) \geq \mathrm{end}(\sigma_s)$, which implies that $\sigma_s \cdot w \in \mathrm{tw}(\Sigma)$. Since $w \in \mathrm{CanD}(\sigma_c')$, we have $\mathrm{start}(w) \geq t$ and $w \succcurlyeq_d \sigma_c'$, which entails that $\Pi_\Sigma(w) = \Pi_\Sigma(\sigma_c')$. Moreover, from $\mathrm{start}(w) \geq t$, we know that all dates of the events in w are greater than or equal to those of the events in $\sigma \cdot (t,a)$. Since i) $\sigma_c \cdot (t,a) = \sigma_c'$, ii) w and σ_c' have the same untimed projection (i.e., $\Pi_\Sigma(w) = \Pi_\Sigma(\sigma_c')$), and iii) the concatenated untimed projections of σ_s and σ_c' form a subword of the untimed projection of $\sigma \cdot (t,a)$ (i.e., $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c') \lhd \Pi_\Sigma(\sigma \cdot (t,a))$), and hence we deduce $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(w) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$. Thus, using $\sigma_s \lhd_d \sigma$ (from induction hypothesis), we obtain $\sigma_s \cdot w \lhd_d \sigma \cdot (t,a) = E_\varphi(\sigma') \lhd_d \sigma'$, i.e., E_φ satisfies $(\textbf{Tr})_{\sigma'}$.
- Case $\kappa_\varphi(\sigma_s, \sigma_c') = \emptyset$. Note, this case encompasses the two last cases in function store_φ. From the definition of E_φ, in both cases we have $E_\varphi(\sigma \cdot (t,a)) = \Pi_1\big(\mathrm{store}_\varphi(\sigma \cdot (t,a))\big) = \sigma_s$. Since $E_\varphi(\sigma) = \Pi_1\big(\mathrm{store}_\varphi(\sigma)\big) = \sigma_s$, and using the induction hypothesis $E_\varphi(\sigma) \models \varphi$, we deduce that $E_\varphi(\sigma') \models \varphi$ $(\textbf{Snd})_{\sigma'}$.
  Moreover, $E_\varphi(\sigma \cdot (t,a)) \lhd_d \sigma$ and thus $E_\varphi(\sigma \cdot (t,a)) \lhd_d \sigma \cdot (t,a)$. We deduce $(\textbf{Tr})_{\sigma'}$.
  Finally, from the induction hypothesis $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$, we can conclude that $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$, proving (ii) for σ'. □

A.2. Proof of Proposition 2 (p. 15)

The proof of Proposition 2 requires the following lemma related to store_φ which says that, when $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and σ_c is not the empty timed word, there is no sequence delaying a prefix of σ_c, starting after the ending date of σ, and allowing to correct σ.

Lemma 1. *Let us consider $\sigma \in \mathrm{tw}(\Sigma)$, if $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma_c \neq \epsilon$, then*

$$\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi.$$

Proof. The proof is done by induction on $\sigma \in \mathrm{tw}(\Sigma)$.

Induction basis For $\sigma = \epsilon$, we have $\sigma_c = \epsilon$ by definition of store_φ, and the induction basis holds.

Induction step Let us suppose that for some $\sigma \in \mathrm{tw}(\Sigma)$, if $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma_c \neq \epsilon$, then $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$ (induction hypothesis). Let us consider $\sigma \cdot (t,a) \in \mathrm{tw}(\Sigma)$, and let $(\sigma_s', \sigma_c') = \mathrm{store}_\varphi(\sigma \cdot (t,a))$. Following the definition of function store_φ, we distinguish three cases:

- If $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t,a)) \neq \emptyset$, then $\sigma_c' = \epsilon$, and the result holds.
- If $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t,a)) = \emptyset$, we have $\sigma_c' = \sigma_c \neq \epsilon$. Using the induction hypothesis, if $\sigma_c' = \sigma_c \neq \epsilon$, we have: $\forall w \in \text{tw}(\Sigma) : (\text{start}(w) \geq \text{end}(\sigma) \wedge \exists v \in \text{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$, which implies $\forall w \in \text{tw}(\Sigma) : (\text{start}(w) \geq \text{end}(\sigma \cdot (t,a)) \wedge \exists v \in \text{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$, which shows that the property holds again for $\sigma \cdot (t,a)$ since $\sigma_c' = \sigma_c$.
- Otherwise ($\kappa_\varphi(\sigma_s, \sigma_c \cdot (t,a)) = \emptyset$ and $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t,a)) \neq \emptyset$), we have $\sigma_c' = \sigma_c \cdot (t,a)$. Using the induction hypothesis, we have: $\forall w \in \text{tw}(\Sigma) : (\text{start}(w) \geq \text{end}(\sigma) \wedge \exists v \in \text{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$, which implies $\forall w \in \text{tw}(\Sigma) : (\text{start}(w) \geq \text{end}(\sigma \cdot (t,a)) \wedge \exists v \in \text{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$. Since $\kappa_\varphi(\sigma_s, \sigma_c') = \emptyset$, by definition we have $\forall w \in \text{tw}(\Sigma) : (\text{start}(w) \geq \text{end}(\sigma \cdot (t,a)) \wedge w \succcurlyeq_d \sigma_c \cdot (t,a)) \implies \sigma_s \cdot w \notin \varphi$. Combining both predicates, we obtain $\forall w \in \text{tw}(\Sigma) : (\text{start}(w) \geq \text{end}(\sigma \cdot (t,a)) \wedge \exists v \in \text{pref}(\sigma_c \cdot (t,a)) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$. □

Let us now return to the proof of Proposition 2. We shall prove that, given a property φ, the associated enforcement function $E_\varphi : \text{tw}(\Sigma) \to \text{tw}(\Sigma)$ as per Definition 8 (p. 14) satisfies the optimality constraint (**Op**) (from Proposition 2, p. 15). That is, we shall prove that $\forall \sigma \in \text{tw}(\Sigma) : (\mathbf{Op})_\sigma$, where:

$$
\begin{aligned}
(\mathbf{Op})_\sigma \stackrel{\text{def}}{=}\ & E_\varphi(\sigma) = \epsilon \vee \exists m, w \in \text{tw}(\Sigma) : E_\varphi(\sigma) = m \cdot w (\models \varphi), \text{ with} \\
& m_\sigma = \max^{\varphi}_{\prec,\epsilon}(E_\varphi(\sigma)), \text{ and} \\
& w_\sigma = \min_{\preceq_{\text{lex}},\text{end}}\{w' \in m_\sigma^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m_\sigma^{-1} \cdot E_\varphi(\sigma)) \\
& \qquad \wedge m_\sigma \cdot w' \lhd_d \sigma \wedge \text{start}(w') \geq \text{end}(\sigma)\}
\end{aligned}
$$

The proof is done by induction on $\sigma \in \text{tw}(\Sigma)$.

Induction basis Since $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon)$ we get $E_\varphi(\epsilon) = \epsilon$.

Induction step Let us suppose that $(\mathbf{Op})_\sigma$ holds for some $\sigma \in \text{tw}(\Sigma)$ (induction hypothesis). Let us consider $\sigma' = \sigma \cdot (t,a)$ with $t \in \mathbb{R}_{\geq 0}$, $t \geq \text{end}(\sigma)$, and $a \in \Sigma$. Let us prove that $(\mathbf{Op})_{\sigma'}$ holds. Suppose $\text{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma_c' = \sigma_c \cdot (t,a)$. We distinguish two cases depending on whether $\kappa_\varphi(\sigma_s, \sigma_c') \neq \emptyset$ or not:

- Case $\kappa_\varphi(\sigma_s, \sigma_c') \neq \emptyset$. We have $E_\varphi(\sigma \cdot (t,a)) = \Pi_1\big(\text{store}_\varphi(\sigma \cdot (t,a))\big) = \sigma_s \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c')$.
  By definition of $\kappa_\varphi(\sigma_s, \sigma_c')$ we know that $\sigma_s \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c') \in \varphi$. From the definition of function store_φ and the induction hypothesis, we know that σ_s corresponds to $m_{\sigma'}$ in the definition of $(\mathbf{Op})_{\sigma'}$: it is the maximal strict prefix of $E_\varphi(\sigma') = \sigma_s \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c')$ that satisfies φ. Indeed, $\text{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and, either $\sigma_c = \epsilon$, then $E_\varphi(\sigma') = \sigma_s.(t', a)$ for some t' and σ_s is the maximal strict prefix of $E_\varphi(\sigma')$ satisfying φ; or $\sigma_c \neq \epsilon$ and using Lemma 1, we know that none of the prefixes of σ_c can be delayed in such a way that, when appended to σ_s, the concatenation forms a correct sequence.
  It follows that $E_\varphi(\sigma \cdot (t,a)) = m_{\sigma'} \cdot w_{\sigma'}$ with $m_{\sigma'} = \sigma_s$ and

$$
\begin{aligned}
w_{\sigma'} &= \sigma_s^{-1} \cdot E_\varphi(\sigma \cdot (t,a)), \\
&= \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c') \\
&= \min_{\preceq_{\text{lex}},\text{end}} \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \underbrace{\sigma_c \cdot (t,a)}_{\sigma_c'} \wedge \text{start}(w') \geq \text{end}(\sigma_c') \right\}.
\end{aligned}
$$

  Since $\text{end}(\sigma_c') = t$, then

$$
w_{\sigma'} = \min_{\preceq_{\text{lex}},\text{end}} \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \sigma_c \cdot (t,a) \wedge \text{start}(w') \geq t \right\}.
$$

  We shall prove that

$$
\begin{aligned}
& \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \sigma_c \cdot (t,a) \wedge \text{start}(w') \geq t \right\} \\
&= \big\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) \wedge m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a) \\
&\qquad \wedge \text{start}(w') \geq \text{end}(\sigma \cdot (t,a)) \big\},
\end{aligned}
$$

  that is (since $\text{end}(\sigma \cdot (t,a)) = t$):

$$
\begin{aligned}
& \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \sigma_c \cdot (t,a) \wedge \text{start}(w') \geq t \right\} \\
&= \big\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) \wedge m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a) \\
&\qquad \wedge \text{start}(w') \geq t \big\}.
\end{aligned}
$$

This amounts to prove that:

$$\forall w' \in m_{\sigma'}^{-1} \cdot \varphi : \mathrm{start}(w') \geq t \implies \big(w' \succcurlyeq_d \sigma_c \cdot (t,a) \iff (\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a))) \wedge m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a)\big).$$

($\Rightarrow$) Since $\Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) = \Pi_\Sigma(\sigma_c \cdot (t,a))$, by definition of $\succcurlyeq_d$, we have $\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)))$. From transparency, we know that $\sigma_s \lhd_d \sigma$ and $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$. Then, from $\mathrm{start}(w') \geq t$, we deduce $m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a)$.

($\Leftarrow$) From $\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a))$, w' and $m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a) = m_{\sigma'}^{-1} \cdot \sigma_c \cdot (t,a)$ have the same events. Moreover, since $\mathrm{start}(w') \geq t$, all events in w' have greater dates than t (and hence, greater than those of all events in $\sigma_c \cdot (t,a)$). Thus $w' \succcurlyeq_d \sigma_c \cdot (t,a)$.

Thus, we conclude that E_φ satisfies $(\mathbf{Op})_{\sigma'}$.

- Case $\kappa_\varphi(\sigma_s, \sigma_c') = \emptyset$. We have $E_\varphi(\sigma \cdot (t,a)) = \Pi_1\big(\mathrm{store}_\varphi(\sigma \cdot (t,a))\big) = \Pi_1\big(\mathrm{store}_\varphi(\sigma)\big) = \sigma_s = E_\varphi(\sigma)$. Thus, from the induction hypothesis, we deduce that $(\mathbf{Op})_{\sigma'}$ holds. □

A.3. Preliminaries to the proof of Proposition 3 (p. 24): characterising the configurations of enforcement monitors

We define some notions and lemmas related to the configurations of any enforcement monitor $\mathcal{E}$.

Remark 9. In the following proofs, without loss of generality, we assume that at any date, in addition to rule ***idle***, at most one of the ***store*** and ***release*** rules of the enforcement monitor applies. This simplification does not come at the price of reducing the generality nor the validity of the proofs because i) rules ***store*** and ***release*** of the enforcement monitor do not rely on the same conditions, and ii) the ***store*** and ***release*** operations of enforcement monitors are assumed to be executed in zero time. The considered simplification however reduces the number of (equivalent) cases in the following proofs.

Remark 10. Between the occurrences of two (input or output) events, the configuration of the enforcement monitor evolves according to rule ***idle*** (since it is the rule with lowest priority). Moreover, from any configuration, applying ***idle*** twice consecutively each delaying for δ_1 and δ_2, or applying ***idle*** once from the same configuration, with delay $\delta_1 + \delta_2$ will result in the same configuration. To simplify notations we will use a rule to simplify the representation of $\mathcal{E}^{\mathrm{ioo}} \in \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ stating that

$$\sigma \cdot (\epsilon, \mathrm{idle}(\delta_1), \epsilon) \cdot (\epsilon, \mathrm{idle}(\delta_2), \epsilon) \cdot \sigma' \text{ is equivalent to } \sigma \cdot (\epsilon, \mathrm{idle}(\delta_1 + \delta_2), \epsilon) \cdot \sigma',$$

for any $\sigma, \sigma' \in \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ and $\delta_1, \delta_2 \in \mathbb{R}_{\geq 0}$. Thus, for $\mathcal{E}^{\mathrm{ioo}}$, we will only consider sequences of $\big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ where delays appearing in operation idle are maximal (i.e., there is no sequence of two consecutive events with an idle operation).

A.3.1. Some notations

Based on the assumption stated in Remark 9, there are at most two configurations for each date. Let us define the two functions $\mathrm{config}_{\mathrm{in}}, \mathrm{config}_{\mathrm{out}} : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to C^{\mathcal{E}}$ that give respectively the first and last configurations of an enforcement monitor at some time instant, reading an input sequence. More formally, given some $\sigma \in \mathrm{tw}(\Sigma), t \in \mathbb{R}_{\geq 0}$:

- $\mathrm{config}_{\mathrm{in}}(\sigma, t) = c_\sigma^t$ such that $c_0^{\mathcal{E}} \overset{w(\sigma,t)}{\hookrightarrow}{}^*_{\mathcal{E}}\ c_\sigma^t$ where $w(\sigma,t) \overset{\mathrm{def}}{=} \min_{\preceq}\{w \preceq \mathcal{E}^{\mathrm{ioo}}(\sigma,t) \mid \mathrm{timeop}(w) = t\}$;
- $\mathrm{config}_{\mathrm{out}}(\sigma, t) = c_\sigma^t$ such that $c_0^{\mathcal{E}} \overset{\mathcal{E}^{\mathrm{ioo}}(\sigma,t)}{\hookrightarrow}{}^*_{\mathcal{E}}\ c_\sigma^t$.

Observe that, when at some date, only rule ***idle*** applies, $\mathrm{config}_{\mathrm{in}}(\sigma,t) = \mathrm{config}_{\mathrm{out}}(\sigma,t)$ holds, because there is only one configuration at this date. Moreover, when at some date, other rules apply (rules ***release*** or ***store***), $\mathrm{config}_{\mathrm{in}}(\sigma,t)$ and $\mathrm{config}_{\mathrm{out}}(\sigma,t)$ differ. Note, in all cases, from $\mathrm{config}_{\mathrm{out}}(\sigma,t)$ only rule ***idle*** applies (which increases time).

Moreover, for any $\sigma \in \mathrm{tw}(\Sigma)$, for any two $t, t' \in \mathbb{R}_{\geq 0}$ such that $t \leq t'$, we note $\mathcal{E}(\sigma,t,t')$ for $\mathcal{E}(\sigma,t)^{-1} \cdot \mathcal{E}(\sigma,t')$, i.e., the output sequence of an enforcement monitor between t and t'. Note that, when $t = t'$, we have $\mathcal{E}(\sigma,t,t') = \epsilon$, for any $\sigma \in \mathrm{tw}(\Sigma)$.

The following remark states that configurations keep track of global time, and is a direct consequence of the rules of enforcement monitors in Definition 10 (p. 21).

Remark 11 *(Value of the third component of configurations).* Only rule ***idle*** modifies the value of the third component of configurations: it increments the third component as time elapses. That is:

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \Pi_3\left(\mathrm{config}_{\mathrm{in}}(\sigma,t)\right) = \Pi_3(\mathrm{config}_{\mathrm{out}}(\sigma,t)) = t.$$

A.3.2. Some intermediate lemmas

Before tackling the proof of Proposition 3, we give a list of lemmas that describe the behaviour of an enforcement monitor, describing the configurations or the output at some particular date for some input and memory content.

Similarly to the first physical constraint, the following lemma states that the enforcement monitor cannot change what it has output. More precisely, when the enforcement monitor is seen as function $\mathcal{E}$, the output is monotonic w.r.t. $\preceq$.

Lemma 2 *(Monotonicity of enforcement monitors). Function* $\mathcal{E} : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to \mathrm{tw}(\Sigma)$ *is monotonic in its second parameter:*

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t, t' \in \mathbb{R}_{\geq 0} : t \leq t' \implies \mathcal{E}(\sigma, t) \preceq \mathcal{E}(\sigma, t').$$

The lemma states that for any input sequence σ, if we consider two dates t, t' such that $t \leq t'$, then the output of the enforcement monitor at date t is a prefix of the output at date t'.

Proof of Lemma 2. The proof directly follows from the definitions of the function $\mathcal{E}$ associated to an enforcement monitor (see Section 7.4, p. 22) which directly depends on $\mathcal{E}^{\mathrm{ioo}}$, which is itself monotonic over time (because of the definition of enforcement monitors). □

As a consequence, one can naturally split the output of the enforcement monitor over time, as it is stated by the following corollary.

Lemma 3 *(Separation of the output of the enforcement monitor over time).*

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t_1, t_2, t_3 \in \mathbb{R}_{\geq 0} : t_1 \leq t_2 \leq t_3 \implies \mathcal{E}(\sigma, t_1, t_3) = \mathcal{E}(\sigma, t_1, t_2) \cdot \mathcal{E}(\sigma, t_2, t_3).$$

The lemma states that for any sequence σ input to $\mathcal{E}$, if we consider three dates $t_1, t_2, t_3 \in \mathbb{R}_{\geq 0}$ such that $t_1 \leq t_2 \leq t_3$, the output of $\mathcal{E}$ between t_1 and t_3 is the concatenation of the output between t_1 and t_2 and the output between t_2 and t_3.

Proof of Lemma 3. Recall that for any $t, t' \in \mathbb{R}_{\geq 0}$ such that $t \leq t'$, $\mathcal{E}(\sigma, t, t')$ is the output sequence of an enforcement monitor between t and t'. The lemma directly follows from the definition of $\mathcal{E}(\sigma, t, t') = \mathcal{E}(\sigma, t)^{-1} \cdot \mathcal{E}(\sigma, t')$. □

The following lemma states that, at some date t, the output of the enforcement monitor only depends on what has been observed until date t. In other words, the enforcement monitor works in an online fashion.

Lemma 4 *(Dependency of the output on the observation only).*

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathcal{E}(\sigma, t) = \mathcal{E}(\mathrm{obs}(\sigma, t), t).$$

Proof of Lemma 4. The proof of the lemma directly follows from the definitions of $\mathcal{E}^{\mathrm{ioo}}$ (Definition 11, p. 22) and obs (in Section 3). Indeed, using $\mathrm{obs}(\sigma, t) = \mathrm{obs}(\mathrm{obs}(\sigma, t), t)$, we deduce that $\mathcal{E}^{\mathrm{ioo}}(\sigma, t) = \mathcal{E}^{\mathrm{ioo}}(\mathrm{obs}(\sigma, t), t)$, for any $\sigma \in \mathrm{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$. Using $\mathcal{E}(\sigma, t) = \Pi_3(\mathcal{E}^{\mathrm{ioo}}(\sigma, t))$, we can deduce the expected result. □

The following lemma states that after reading some input sequence σ entirely, only the memory content σ_{ms} and the value of the clock t influence the output of the enforcement monitor. More specifically, after completely reading some sequence, if an enforcement monitor reaches some configuration containing σ_{ms} in its memory, its future output is fully determined by the memory content σ_{ms} (containing the corrected sequence) and the value of clock variable t, during the total time needed to output it.

Lemma 5 *(Values of* $\mathrm{config}_{\mathrm{out}}$ *when releasing events).*

$$\begin{aligned}
&\forall \sigma, \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma), \forall t, t_F \in \mathbb{R}_{\geq 0}, \forall q \in Q : \\
&\quad t \geq \mathrm{end}(\sigma) \wedge \mathrm{config}_{\mathrm{out}}(\sigma, t) = (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \\
&\quad \implies \forall \sigma'_{\mathrm{ms}} \preceq \sigma_{\mathrm{ms}} : \mathrm{config}_{\mathrm{out}}(\sigma, \mathrm{end}(\sigma'_{\mathrm{ms}})) = (\sigma'^{-1}_{\mathrm{ms}} \cdot \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, \mathrm{end}(\sigma'_{\mathrm{ms}}), q, t_F).
\end{aligned}$$

The lemma states that, whatever is the output configuration $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F)$ reached by reading some input sequence σ at some date $t \geq \mathrm{end}(\sigma)$, then for any prefix σ'_{ms} of σ_{ms}, the output configuration reached at time $\mathrm{end}(\sigma'_{\mathrm{ms}})$ (output date of the last event in σ'_{ms}) is such that σ'_{ms} has been released from the memory (the memory is thus $\sigma'^{-1}_{\mathrm{ms}} \cdot \sigma_{\mathrm{ms}}$) and the clock value in this configuration is $\mathrm{end}(\sigma'_{\mathrm{ms}})$.

Proof of Lemma 5. The proof is a straightforward induction on the length of σ'_{ms}. It uses the fact that the considered configurations occur at dates greater than $\mathrm{end}(\sigma)$, hence implying that no input event can be read any more. Consequently,

following the definition of the enforcement monitor (Definition 10, p. 21), on the configurations of the enforcement monitor, only rules ***idle*** and ***release*** apply. Between $\text{end}(\sigma'_{\text{ms}})$ and $\text{end}(\sigma'_{\text{ms}} \cdot (t,a))$ where $\sigma'_{\text{ms}} \preceq \sigma'_{\text{ms}} \cdot (t,a) \preceq \sigma_{\text{ms}}$, the configuration of the enforcement monitor evolves only using rule ***idle*** (no other rule applies) until $\text{config}_{\text{in}}(\sigma, \text{end}(\sigma'_{\text{ms}} \cdot (t,a))) = (\sigma'^{-1}_{\text{ms}} \cdot \sigma_{\text{ms}}, \sigma_{\text{mc}}, \text{end}(\sigma'_{\text{ms}} \cdot (t,a)), q, t_F)$. Rule ***release*** is then applied to get the following derivation $(\sigma'^{-1}_{\text{ms}} \cdot \sigma_{\text{ms}}, \sigma_{\text{mc}}, \text{end}(\sigma'_{\text{ms}} \cdot (t,a)), q) \overset{\epsilon/\text{release}(t,a)/\epsilon}{\hookrightarrow} ((\sigma'_{\text{ms}} \cdot (t,a))^{-1} \cdot \sigma_{\text{ms}}, \sigma_{\text{mc}}, \text{end}(\sigma'_{\text{ms}} \cdot (t,a)), q, t_F)$. □

The following lemma relates the date of the last event of the corrected sequence and the value of the last variable stored in the configuration of an enforcement monitor.

Lemma 6 *(Relation between some elements in a configuration).*

$$\forall \sigma, \sigma_{\text{ms}} \in \text{tw}(\Sigma), \forall t, t_F \in \mathbb{R}_{\geq 0} : \text{config}_{\text{out}}(\sigma, t) = (\sigma_{\text{ms}}, \_, t, \_, t_F) \wedge \sigma_{\text{ms}} \neq \epsilon \implies \text{end}(\sigma_{\text{ms}}) = t_F.$$

Proof. The lemma is a straightforward consequence of the definition of enforcement monitors (Definition 10, p. 21). Indeed, only rule *store-*φ modifies these elements of a configuration, and it performs it as expected. □

The following lemma states that when an enforcement monitor has nothing to read in input anymore, what it releases as output is the observation of its memory content over time.

Lemma 7 *(Output of the enforcement monitor according to memory content).*

$$\begin{aligned}&\forall \sigma, \sigma_{\text{ms}}, \sigma_{\text{mc}} \in \text{tw}(\Sigma), \forall t, t_F \in \mathbb{R}_{\geq 0}, \forall q \in Q :\\ &\quad t \geq \text{end}(\sigma) \wedge \text{config}_{\text{out}}(\sigma, t) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q, t_F)\\ &\qquad \implies \forall t' \in \mathbb{R}_{\geq 0} : t \leq t' \leq \text{end}(\sigma_{\text{ms}}) \implies \mathcal{E}(\sigma, t, t') = \text{obs}(\sigma_{\text{ms}}, t').\end{aligned}$$

The lemma states that, if after some date t, after reading an input sequence σ, the enforcement monitor is in an output configuration that contains σ_{ms} as a memory content, whatever is the date t' between t and $\text{end}(\sigma_{\text{ms}})$, the output of the enforcement monitor between t and t' is the observation of σ_{ms} with t' time units.

Proof of Lemma 7. The proof is performed by induction on the length of σ_{ms} and uses Lemma 5.

- Case $|\sigma_{\text{ms}}| = 0$. In this case, $\sigma_{\text{ms}} = \epsilon$ and $\text{end}(\epsilon) = 0$. If $t = t' = 0$, we have $\mathcal{E}(\sigma, t, t') = \epsilon = \text{obs}(\sigma_{\text{ms}}, t')$. Otherwise, $t \leq t'$ does not hold, and thus the lemma vacuously holds.
- Induction case. Let us suppose that the lemma holds for all prefixes of σ_{ms} of some maximum length $n \in [0, |\sigma_{\text{ms}}| - 1]$ (induction hypothesis). Following Lemma 6, one can consider $\sigma_{\text{ms}} = \sigma' \cdot (t_F, a)$ where σ' is the prefix of σ_{ms} of length n, and $(t_F, a) \in \mathbb{R}_{\geq 0} \times \Sigma$. On the one hand, at date $\text{end}(\sigma')$, according to Lemma 5, we have $\text{config}_{\text{out}}(\sigma, \text{end}(\sigma')) = ((t_F, a), \sigma_{\text{mc}}, \text{end}(\sigma'), q, t_F)$ for some $\sigma_{\text{mc}} \in \text{tw}(\Sigma)$ and $q \in Q$. For any $t' \leq \text{end}(\sigma')$, the lemma vacuously holds. On the other hand, let us consider some $t' \in [\text{end}(\sigma'), t_F]$, we have:

  $$\mathcal{E}(\sigma, t, t') = \mathcal{E}(\sigma, t, \text{end}(\sigma')) \cdot \mathcal{E}(\sigma, \text{end}(\sigma'), t').$$

  (Note, when $t = t' = \text{end}(\sigma')$, the above equation reduces to $\epsilon = \epsilon$.) Using the induction hypothesis, we find $\mathcal{E}(\sigma, t, \text{end}(\sigma')) = \text{obs}(\sigma', \text{end}(\sigma')) = \sigma'$. Using the semantics of the enforcement monitor (only rules ***release*** and ***idle*** apply, no new event is received), we obtain $\mathcal{E}(\sigma, \text{end}(\sigma'), t') = \text{obs}((t_F, a), t')$. Thus, $\mathcal{E}(\sigma, t, t') = \sigma' \cdot \text{obs}((t_F, a), t') = \text{obs}(\sigma' \cdot (t_F, a), t')$. □

The following lemma states that, for any input σ, after observing the entire input (that is, at any date greater than or equal to $\text{end}(\sigma)$), the content of the internal memory (σ_c) of the enforcement function and the enforcement monitor are the same.

Lemma 8 *(Content of the internal memory).*

$$\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : t \geq \text{end}(\sigma) \implies \Pi_2(\text{store}_{\varphi}(\sigma)) = \Pi_2(\text{config}_{\text{out}}(\sigma, t)).$$

Proof of Lemma 8. The proof is performed by induction on the length of σ. Recall that $\text{store}_{\varphi}(\sigma)$ is defined in Section 6.2, and $\text{config}_{\text{out}}(\sigma, t)$ is defined in Appendix A.3.1.

- Case $|\sigma| = 0$. In this case, from the definition of the enforcement monitor (Definition 10, p. 21), none of the store rules can be applied. Consequently, we have $\Pi_2(\text{config}_{\text{out}}(\sigma, t)) = \epsilon$. Regarding the enforcement function, as per Definition 8 (p. 14), we have $\Pi_2(\text{store}_{\varphi}(\epsilon)) = \epsilon$.

- Induction case. Let us suppose that for some $\sigma \in \mathrm{tw}(\Sigma)$, we have $\forall t \in \mathbb{R}_{\geq 0} : t \geq \mathrm{end}(\sigma) \implies \Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t))$ (induction hypothesis). Let us consider $\sigma' = \sigma \cdot (t_l, a)$, where $(t_l, a) \in \mathbb{R}_{\geq 0} \times \Sigma$.
  From the induction hypothesis, for $t \geq \mathrm{end}(\sigma)$, we have $\Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t))$, and therefore, for any $t \geq t_l$, we also have $\Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t))$. Let $\sigma_c = \Pi_2(\mathrm{store}_\varphi(\sigma))$. Consequently, we also have $\mathrm{config}_{\mathrm{in}}(\sigma, t_l) = (\_, \sigma_c, t_l, \_, \_) = \mathrm{config}_{\mathrm{in}}(\sigma \cdot (t_l, a), t_l)$.
  From the definition of store_φ, we have $\Pi_2(\mathrm{store}_\varphi(\sigma \cdot (t_l, a))) = \sigma_c'$, where σ_c' is either ϵ, $\sigma_c \cdot (t_l, a)$, or σ_c depending on which case of the store_φ function applies.
  Regarding the enforcement monitor, from the update function (since each case in store_φ has a corresponding case in update), we also have $\mathrm{config}_{\mathrm{out}}(\sigma \cdot (t_l, a), t_l) = (\_, \sigma_c', t_l, \_, \_)$ (which is obtained by applying one of the store rules based on the value returned by function update). For $t > t_l$, since none of the store rules can be applied, we can conclude that $\mathrm{config}_{\mathrm{out}}(\sigma \cdot (t_l, a), t) = (\_, \sigma_c', t, \_, \_)$.
  Thus, we have $\Pi_2(\mathrm{store}_\varphi(\sigma \cdot (t_l, a))) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma \cdot (t_l, a), t))$. □

A.4. Proof of Proposition 3: relation between enforcement function and enforcement monitor

We shall prove that, given a property φ, the associated enforcement monitor $\mathcal{E}_\varphi$ as per Definition 10 (p. 21) implements the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ as per Definition 8 (p. 14). That is:

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\sigma, t).$$

The proof is done by induction on the length of the input timed word σ.

Induction basis Let us suppose that $|\sigma| = 0$, thus $\sigma = \epsilon$ in $\mathrm{tw}(\Sigma)$. On the one hand, we have $E_\varphi(\sigma) = \epsilon$, and thus $\forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma), t) = \epsilon$. On the other hand, the word $\mathcal{E}_\varphi^{\mathrm{ioo}}(\epsilon, t)$ over the input-operation-output alphabet is such that $\forall t \in \mathbb{R}_{\geq 0} : \Pi_1(\mathcal{E}_\varphi^{\mathrm{ioo}}(\epsilon, t)) = \epsilon$. Thus, according to the definition of the enforcement monitor, the rules ***store-***φ, ***store***$_{\mathrm{sup}}$***-***$\overline{\varphi}$, and ***store-***$\overline{\varphi}$ cannot be applied. Consequently, the memory of the enforcement monitor σ_{ms} remains empty as in the initial configuration. It follows that rule ***release*** cannot be applied as well. We have then $\forall t \in \mathbb{R}_{\geq 0} : c_0^{\mathcal{E}_\varphi} \overset{\epsilon/\mathrm{idle}(t)/\epsilon}{\hookrightarrow} \mathcal{E}_\varphi (\epsilon, \epsilon, t, q_0, 0)$, and thus $\mathcal{E}_\varphi(\epsilon, t) = \epsilon$. Thus, $\forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\epsilon, t)$.

Induction step Let us suppose that $\mathrm{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\sigma, t)$ for any timed word $\sigma \in \mathrm{tw}(\Sigma)$ of some length $n \in \mathbb{N}$, at any date $t \in \mathbb{R}_{\geq 0}$ (induction hypothesis). Let us now consider some input timed word $\sigma \cdot (t_{n+1}, a)$ for some $\sigma \in \mathrm{tw}(\Sigma)$ with $|\sigma| = n$, $t_{n+1} \in \mathbb{R}_{\geq 0}$, and $a \in \Sigma$. We want to prove that $\mathrm{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t)$, at any date $t \in \mathbb{R}_{\geq 0}$.

Let us consider some date $t \in \mathbb{R}_{\geq 0}$. Note that $\mathrm{end}(\sigma \cdot (t_{n+1}, a)) = t_{n+1}$. We distinguish two cases according to whether $t_{n+1} > t$ or not, that is whether $\sigma \cdot (t_{n+1}, a)$ is completely observed or not at date t.

- Case $t_{n+1} > t$. In this case, $\mathrm{obs}(\sigma \cdot (t_{n+1}, a), t) = \mathrm{obs}(\sigma, t)$, i.e., at date t, the observations of σ and $\sigma \cdot (t_{n+1}, a)$ are identical.
  On the one hand, from the definition of E_φ (since function store_φ and the delayed subsequence are defined such that the date of each event in output is greater than or equal to the date of the corresponding event in the input), we have:

$$\begin{aligned} \mathrm{obs}\left(E_\varphi\left(\sigma \cdot (t_{n+1}, a)\right), t\right) &= \mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi(\sigma \cdot (t_{n+1}, a))\right), t\right) \\ &= \mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi(\sigma)\right), t\right) \\ &= \mathrm{obs}\left(E_\varphi\left(\sigma\right), t\right). \end{aligned}$$

  On the other hand, regarding the enforcement monitor, since $\mathrm{obs}\left(\sigma \cdot (t_{n+1}, a), t\right) = \mathrm{obs}(\sigma, t)$, using Lemma 4 (p. 35), we obtain $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}_\varphi(\sigma, t)$. Using the induction hypothesis, we can conclude that $\mathrm{obs}\left(E_\varphi\left(\sigma \cdot (t_{n+1}, a)\right), t\right) = \mathcal{E}_\varphi\left(\sigma \cdot (t_{n+1}, a), t\right)$.
- Case $t_{n+1} \leq t$. In this case, we have $\mathrm{obs}(\sigma \cdot (t_{n+1}, a), t) = \sigma \cdot (t_{n+1}, a)$ (i.e., $\sigma \cdot (t_{n+1}, a)$ is observed entirely at date t). From Remark 11 (p. 34), we know that the configuration of the enforcement monitor at date $\mathrm{end}(\sigma \cdot (t_{n+1}, a))$ is $\mathrm{config}_{\mathrm{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right) = (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t_{n+1}, q_\sigma, t_F)$ for some $\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma)$, $q_\sigma \in Q$, $t_F \in \mathbb{R}_{\geq 0}$. Using Lemma 8 (p. 36), we also have $\Pi_2(\mathrm{store}_\varphi(\sigma)) = \sigma_c = \Pi_2(\mathrm{config}_{\mathrm{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right)) = \sigma_{\mathrm{mc}}$. Observe that $\mathrm{config}_{\mathrm{in}}\left(\sigma, t_{n+1}\right) = \mathrm{config}_{\mathrm{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right)$ because of i) the definition of $\mathrm{config}_{\mathrm{in}}$ using the definition of $\mathcal{E}_\varphi^{\mathrm{ioo}}$ and ii) the event (t_{n+1}, a) has not been yet consumed through any of the ***store*** rules by the enforcement monitor at date t_{n+1}.
  We distinguish two cases according to whether $\sigma_c \cdot (t_{n+1}, a)$ can be delayed into a word satisfying φ or not, i.e., whether $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) = \emptyset$, or not.
  - Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) = \emptyset$. From the definition of function store_φ, we have $\mathrm{store}_\varphi(\sigma \cdot (t_{n+1}, a)) = (\sigma_s, \sigma_c')$, and $\Pi_1\left(\mathrm{store}_\varphi(\sigma \cdot (t_{n+1}, a))\right) = \sigma_s$. We also have $\Pi_1\left(\mathrm{store}_\varphi(\sigma)\right) = \sigma_s$. From the definition of E_φ and obs, we have $\mathrm{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \mathrm{obs}(E_\varphi(\sigma), t)$.
    Regarding $\mathcal{E}_\varphi$, according to the definition of function update, we have $\mathrm{update}(q_\sigma, t_F, \sigma_{\mathrm{mc}}, (t_{n+1}, a)) = (q_\sigma, \sigma_{\mathrm{mc}}, \mathtt{bad})$ or $(q_\sigma, \sigma_{\mathrm{mc}} \cdot (t_{n+1}, a), \mathtt{c\_bad})$. According to the definition of the transition relation, we have:

$$(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma, t_F) \overset{(t_{n+1},a)/\text{store}-\overline{\varphi}(t_{n+1},a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma, t_F),$$

where, $\sigma'_{\text{mc}} = \sigma_{\text{mc}}$ if $\text{update}(q_\sigma, t_F, \sigma_{\text{mc}}, (t_{n+1}, a)) = (q_\sigma, \sigma_{\text{mc}}, \texttt{bad})$, and $\sigma'_{\text{mc}} = \sigma_{\text{mc}} \cdot (t_{n+1}, a)$ otherwise. Thus $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$.
Let us consider $t_\epsilon \in \mathbb{R}_{\geq 0}$ such that between $t_{n+1} - t_\epsilon$ and t_{n+1}, the enforcement monitor does not read any input nor produce any output, i.e., for all $t \in [t_{n+1} - t_\epsilon, t_{n+1}]$, config$(t)$ is such that only the rule ***idle*** applies.
Let us examine $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t)$. We have:

$$\begin{aligned}\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \ & \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) \\ & \cdot \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) \\ & \cdot \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t).\end{aligned}$$

Let us examine $\mathcal{E}_\varphi(\sigma, t)$. We have:

$$\mathcal{E}_\varphi(\sigma, t) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon, t_{n+1}) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1}, t).$$

Observe that $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon)$ because $\text{obs}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \sigma$ according to the definition of obs. Moreover, $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) = \epsilon$ since only rule ***idle*** applies during the considered time interval. Furthermore, according to Lemma 7, since $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$, we get $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}}, t)$. Moreover, we know that $\text{config}_{\text{in}}(\sigma, t_{n+1}) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$. Since the enforcement monitor is deterministic, and from Remark 9 (p. 34), we also get that $\text{config}_{\text{out}}(\sigma, t_{n+1}) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$. Using Lemma 7 (p. 36) again, we get $\mathcal{E}_\varphi(\sigma, t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}}, t)$.
Consequently we can deduce that $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}_\varphi(\sigma, t) = \text{obs}(E_\varphi(\sigma), t) = \text{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t)$.

– Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) \neq \emptyset$. Regarding E_φ, from the definition of function store_φ, we have $\text{store}_\varphi(\sigma \cdot (t_{n+1}, a)) = (\sigma_s \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)), \epsilon)$, and $\Pi_1\big(\text{store}_\varphi(\sigma \cdot (t_{n+1}, a))\big) = \sigma_s \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a))$. Regarding the enforcement monitor, according to the definition of update, we have $\text{update}(q_\sigma, \sigma_{\text{mc}}, (t_{n+1}, a), t_F) = (q', w, \texttt{ok})$ with $w = \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a))$, since, $\sigma_c = \sigma_{\text{mc}}$ and from the definition of κ_φ and update, the dates computed for $\sigma_c \cdot (t_{n+1}, a)$ by both these functions are equal. From the definition of the transition relation, we have:

$$(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma, t_F) \overset{(t_{n+1},a)/\text{store}-\varphi(t_{n+1},a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q', \text{end}(w)).$$

Thus $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q', \text{end}(w))$.
Let us consider $t_\epsilon \in \mathbb{R}_{\geq 0}$ such that between $t_{n+1} - t_\epsilon$ and t_{n+1}, the enforcement monitor does not read any input nor produce any output, i.e., for all $t \in [t_{n+1} - t_\epsilon, t_{n+1}]$, config$(t)$ is such that only rule ***idle*** applies.
Let us examine $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t)$. We have:

$$\begin{aligned}\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \ & \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) \\ & \cdot \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) \\ & \cdot \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t).\end{aligned}$$

Let us examine $\mathcal{E}_\varphi(\sigma, t)$. We have:

$$\mathcal{E}_\varphi(\sigma, t) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon, t_{n+1}) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1}, t).$$

Observe that $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon)$ because $\text{obs}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \sigma$ according to the definition of obs. Moreover, $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) = \epsilon$ since only rule ***idle*** applies during the considered time interval.
Furthermore, according to Lemma 7 (p. 36), since $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q', \text{end}(w))$, we get $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}} \cdot w, t)$.
Now we further distinguish two more sub-cases, based on whether $\text{end}(\sigma_{\text{ms}} \cdot w) = \text{end}(w) > t$ or not (whether all the elements in the memory can be released as output by date t or not).

∗ Case $\text{end}(w) > t$.
We further distinguish two more sub-cases based on whether $\text{end}(\sigma_{\text{ms}}) > t$, or not.

· Case $\text{end}(\sigma_{\text{ms}}) > t$. In this case, we know that $\text{obs}(\sigma_{\text{ms}} \cdot w, t) = \text{obs}(\sigma_{\text{ms}}, t)$. Hence, we can derive that $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}_\varphi(\sigma, t)$. Also, from the induction hypothesis, we know that $\mathcal{E}_\varphi(\sigma, t) = \text{obs}(E_\varphi(\sigma), t)$.
Regarding enforcement function E_φ, we have

$$\text{store}_\varphi(\sigma \cdot (t_{n+1}, a)) = \Pi_1\big(\text{store}_\varphi(\sigma)\big) \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)).$$

Moreover,

$$\begin{aligned}&\text{obs}\left(E_\varphi\left(\sigma \cdot (t_{n+1}, a)\right), t\right) \\ &\quad = \text{obs}\left(\Pi_1\left(\text{store}_\varphi\left(\sigma \cdot (t_{n+1}, a)\right)\right), t\right) \\ &\quad = \text{obs}\left(\Pi_1\left(\text{store}_\varphi\left(\sigma\right)\right) \cdot \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)), t\right).\end{aligned}$$

One can have

$$\text{obs}\left(E_\varphi\left(\sigma\cdot(t_{n+1},a)\right),t\right)=\Pi_1\left(\text{store}_\varphi(\sigma)\right)\cdot o,$$

where $o \preccurlyeq \min_{\preceq_{\text{lex}},\text{end}}\ \kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a))$, which is equal to $\text{obs}(E_\varphi(\sigma),t)\cdot o$, only if the dates computed by the update function are different from the dates computed by E_φ. This would violate the induction hypothesis stating that $\mathcal{E}_\varphi(\sigma,t)=\text{obs}(E_\varphi(\sigma),t)$. Hence, we have $\text{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\text{obs}\left(\Pi_1\left(\text{store}_\varphi\left(\sigma\right)\right),t\right)=\text{obs}(E_\varphi(\sigma),t)$. Thus, $\text{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)$.

- Case $\text{end}(\sigma_{\text{ms}})\leq t$. In this case, we can follow the same reasoning as in the previous case to obtain the expected result.

∗ Case $\text{end}(w)\leq t$.
In this case, similarly following Lemma 7 (p. 36), we have $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t_{n+1},t)=\text{obs}(\sigma_{\text{ms}}\cdot w,t)=\sigma_{\text{ms}}\cdot w$. We can also derive that $\mathcal{E}_\varphi(\sigma,t_{n+1},t)=\sigma_{\text{ms}}$. Consequently, we have $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)=\mathcal{E}_\varphi(\sigma,t)\cdot w$. From the induction hypothesis, we know that $\text{obs}(E_\varphi(\sigma),t)=\mathcal{E}_\varphi(\sigma,t)$, and we have $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)=\text{obs}(E_\varphi(\sigma),t)\cdot w$.
Moreover, we have

$$\text{store}_\varphi(\sigma\cdot(t_{n+1},a))=\Pi_1\left(\text{store}_\varphi(\sigma)\right)\cdot\min_{\preceq_{\text{lex}},\text{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a)),$$

and thus

$$\begin{aligned}&\text{obs}\left(E_\varphi\left(\sigma\cdot(t_{n+1},a)\right),t\right)\\&\quad=\text{obs}\left(\Pi_1\left(\text{store}_\varphi\left(\sigma\right)\right)\cdot\min_{\preceq_{\text{lex}},\text{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a)),t\right).\end{aligned}$$

Henceforth, we have $\text{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\text{store}_\varphi(\sigma)\cdot\min_{\preceq_{\text{lex}},\text{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a))=E_\varphi(\sigma)\cdot\min_{\preceq_{\text{lex}},\text{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a))$, since, $\sigma_c=\sigma_{\text{mc}}$ and from the definition of κ_φ and update, we know the dates computed for the subsequence $\sigma_c\cdot(t_{n+1},a)$ by E_φ and $\mathcal{E}_\varphi$ are equal. Finally, we have $\text{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)$. □

References

[1] F.B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50, http://dx.doi.org/10.1145/353323.353382.
[2] K.W. Hamlen, G. Morrisett, F.B. Schneider, Computability classes for enforcement mechanisms, ACM Trans. Program. Lang. Syst. 28 (1) (2006) 175–205, http://dx.doi.org/10.1145/1111596.1111601.
[3] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Trans. Inf. Syst. Secur. 12 (3) (2009) 19:1–19:41, http://dx.doi.org/10.1145/1455526.1455532.
[4] Y. Falcone, L. Mounier, J.-C. Fernandez, J.-L. Richier, Runtime enforcement monitors: composition, synthesis, and enforcement abilities, Form. Methods Syst. Des. 38 (3) (2011) 223–262, http://dx.doi.org/10.1007/s10703-011-0114-4.
[5] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, O.L.N. Timo, Runtime enforcement of timed properties, in: S. Qadeer, S. Tasiran (Eds.), Proceedings of the Third International Conference on Runtime Verification, RV 2012, in: Lect. Notes Comput. Sci., vol. 7687, Springer, 2012, pp. 229–244.
[6] B. Zeng, G. Tan, Ú. Erlingsson, Strato: a retargetable framework for low-level inlined-reference monitors, in: S.T. King (Ed.), Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14–16, 2013, USENIX Association, 2013, pp. 369–382.
[7] Ú. Erlingsson, F.B. Schneider, IRM enforcement of Java stack inspection, in: IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14–17, 2000, IEEE Computer Society, 2000, pp. 246–255.
[8] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, Runtime enforcement of parametric timed properties with practical applications, in: J. Lesage, J. Faure, J.E.R. Cury, B. Lennartson (Eds.), 12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14–16, 2014, International Federation of Automatic Control, 2014, pp. 420–427.
[9] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, Runtime enforcement of regular timed properties, in: Y. Cho, S.Y. Shin, S.-W. Kim, C.-C. Hung, J. Hong (Eds.), Proceedings of the ACM Symposium on Applied Computing, SAC-SVT, ACM, 2014, pp. 1279–1286.
[10] Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime?, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 349–382, http://dx.doi.org/10.1007/s10009-011-0196-8.
[11] R. Alur, D.L. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (1994) 183–235, http://dx.doi.org/10.1016/0304-3975(94)90010-8.
[12] J. Bengtsson, W. Yi, Timed automata: semantics, algorithms and tools, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Lectures on Concurrency and Petri Nets, Advances in Petri Nets, in: Lect. Notes Comput. Sci., vol. 3098, Springer, 2003, pp. 87–124.
[13] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, Inf. Comput. 111 (2) (1994) 193–244, http://dx.doi.org/10.1006/inco.1994.1045.
[14] G. Behrmann, A. David, K.G. Larsen, A tutorial on UPPAAL, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13–18, 2004, in: Lect. Notes Comput. Sci., vol. 3185, Springer, 2004, pp. 200–236, Revised lectures.
[15] R. Alur, L. Fix, T.A. Henzinger, Event-clock automata: a determinizable class of timed automata, Theor. Comput. Sci. 211 (1–2) (1999) 253–273, http://dx.doi.org/10.1016/S0304-3975(97)00173-4.
[16] J. Bengtsson, W. Yi, Timed automata: semantics, algorithms and tools, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Proceedings of the 4th Advanced Course on Petri Nets – Lecture Notes on Concurrency and Petri Nets, in: Lect. Notes Comput. Sci., vol. 3098, Springer, 2003, pp. 87–124.
[17] O. Maler, D. Nickovic, A. Pnueli, From MITL to timed automata, in: E. Asarin, P. Bouyer (Eds.), Proceedings of the 4th international conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2006, in: Lect. Notes Comput. Sci., Springer-Verlag, 2006, pp. 274–289.
[18] D. Nickovic, N. Piterman, From MTL to deterministic timed automata, in: K. Chatterjee, T.A. Henzinger (Eds.), Proceedings of the 8th International Conference on Formal Modelling and Analysis of Timed Systems, FORMATS 2010, in: Lect. Notes Comput. Sci., vol. 6246, Springer, 2010, pp. 152–167.
[19] P. Bouyer, T. Brihaye, V. Bruyère, J.-F. Raskin, On the optimal reachability problem of weighted timed automata, Form. Methods Syst. Des. 31 (2) (2007) 135–175, http://dx.doi.org/10.1007/s10703-007-0035-4.
[20] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, TiPEX: a tool chain for timed property enforcement during execution, in: Bartocci and Majumdar [56], pp. 306–320, http://dx.doi.org/10.1007/978-3-319-23820-3_22.
[21] K. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, Int. J. Softw. Tools Technol. Transf. 1 (1–2) (1997) 134–152, http://dx.doi.org/10.1007/s100090050010.

[22] V. Gruhn, R. Laue, Patterns for timed property specifications, Electron. Notes Theor. Comput. Sci. 153 (2) (2006) 117–133, http://dx.doi.org/10.1016/j.entcs.2005.10.035.
[23] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, O. Nguena-Timo, Runtime enforcement of timed properties revisited, Form. Methods Syst. Des. 45 (3) (2014) 381–422, http://dx.doi.org/10.1007/s10703-014-0215-y.
[24] K. Havelund, G. Rosu, Synthesizing monitors for safety properties, in: J. Katoen, P. Stevens (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002„ in: Lect. Notes Comput. Sci., Springer, 2002, pp. 342–356.
[25] K. Havelund, A. Goldberg, Verify your runs, in: B. Meyer, J. Woodcock (Eds.), Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10–13, 2005, in: Lect. Notes Comput. Sci., vol. 4171, Springer, 2005, pp. 374–383, Revised selected papers and discussions.
[26] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Program. 78 (5) (2009) 293–303, http://dx.doi.org/10.1016/j.jlap.2008.08.004.
[27] O. Sokolsky, K. Havelund, I. Lee, Introduction to the special section on runtime verification, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 243–247, http://dx.doi.org/10.1007/s10009-011-0218-6.
[28] Y. Falcone, K. Havelund, G. Reger, A tutorial on runtime verification, in: M. Broy, D. Peled, G. Kalus (Eds.), Engineering Dependable Software Systems, in: NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, IOS Press, 2013, pp. 141–175.
[29] D. Seto, B. Krogh, L. Sha, A. Chutinan, The simplex architecture for safe online control system upgrades, in: Proceedings of the American Control Conference, vol. 6, 1998, pp. 3504–3508.
[30] S. Bak, K. Manamcheri, S. Mitra, M. Caccamo, Sandboxing controllers for cyber-physical systems, in: IEEE/ACM International Conference on Cyber-Physical Systems, ICCPS 2011, Chicago, Illinois, USA, 12–14 April, 2011, IEEE Computer Society, 2011, pp. 3–12.
[31] S. Bak, F.A.T. Abad, Z. Huang, M. Caccamo, Using run-time checking to provide safety and progress for distributed cyber-physical systems, in: IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2013, Taipei, Taiwan, August 19–21, 2013, IEEE, 2013, pp. 287–296.
[32] S. Mitsch, A. Platzer, Modelplex: verified runtime validation of verified cyber-physical system models, in: Bonakdarpour and Smolka [55], pp. 199–214, http://dx.doi.org/10.1007/978-3-319-11164-3_17.
[33] C. Colombo, G.J. Pace, Fast-forward runtime monitoring – an industrial case study, in: S. Qadeer, S. Tasiran (Eds.), Third International Conference on Runtime Verification, RV 2012, Istanbul, Turkey, September 25–28, 2012, in: Lect. Notes Comput. Sci., vol. 7687, Springer, 2012, pp. 214–228, Revised selected papers.
[34] D.A. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, H. Mantel, Scalable offline monitoring, in: Bonakdarpour and Smolka [55], pp. 31–47, http://dx.doi.org/10.1007/978-3-319-11164-3_4.
[35] A. Kassem, Y. Falcone, P. Lafourcade, Monitoring electronic exams, in: Bartocci and Majumdar [56], pp. 118–135, http://dx.doi.org/10.1007/978-3-319-23820-3_8.
[36] A. Goodloe, L. Pike, Monitoring distributed real-time systems: a survey and future directions, Tech. Rep. NASA/CR-2010-216724, NASA Langley Research Center, July 2010, available at http://ntrs.nasa.gov.
[37] O. Sokolsky, U. Sammapun, I. Lee, J. Kim, Run-time checking of dynamic properties, Electron. Notes Theor. Comput. Sci. 144 (4) (2006) 91–108, http://dx.doi.org/10.1016/j.entcs.2006.02.006.
[38] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14:1–14:64, http://dx.doi.org/10.1145/2000799.2000800.
[39] P. Thati, G. Rosu, Monitoring algorithms for metric temporal logic specifications, Electron. Notes Theor. Comput. Sci. 113 (2005) 145–162, http://dx.doi.org/10.1016/j.entcs.2004.01.029.
[40] D.A. Basin, F. Klaedtke, E. Zalinescu, Algorithms for monitoring real-time properties, in: Khurshid and Sen [57], pp. 260–275, http://dx.doi.org/10.1007/978-3-642-29860-8_20.
[41] U. Sammapun, I. Lee, O. Sokolsky, RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties, in: IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, 2005, pp. 147–153.
[42] D. Nickovic, O. Maler, AMT: a property-based monitoring tool for analog systems, in: J.-F. Raskin, P.S. Thiagarajan (Eds.), Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2007, in: Lect. Notes Comput. Sci., vol. 4763, Springer-Verlag, 2007, pp. 304–319.
[43] C. Colombo, G.J. Pace, G. Schneider, Dynamic event-based runtime monitoring of real-time and contextual properties, in: D.D. Cofer, A. Fantechi (Eds.), 13th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2008, L'Aquila, Italy, September 15–16, 2008, in: Lect. Notes Comput. Sci., vol. 5596, Springer, 2008, pp. 135–149, Revised selected papers.
[44] C. Colombo, G.J. Pace, G. Schneider, LARVA – safer monitoring of real-time Java programs (tool paper), in: D.V. Hung, P. Krishnan (Eds.), Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, IEEE Computer Society, 2009, pp. 33–37.
[45] Y. Falcone, You should better enforce than verify, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Proceedings of the First International Conference on Runtime Verification, RV 2010, in: Lect. Notes Comput. Sci., vol. 6418, Springer, 2010, pp. 89–105.
[46] N. Bielova, F. Massacci, Do you really mean what you actually enforced? Edit automata revisited, Int. J. Inf. Secur. 10 (4) (2011) 239–254, http://dx.doi.org/10.1007/s10207-011-0137-2.
[47] P.O. Meredith, G. Rosu, Runtime verification with the RV system, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Proceedings of the First International Conference on Runtime Verification, RV 2010, St. Julians, Malta, November 1–4, 2010, in: Lect. Notes Comput. Sci., vol. 6418, Springer, 2010, pp. 136–152.
[48] C. Colombo, G.J. Pace, P. Abela, Safer asynchronous runtime monitoring using compensations, Form. Methods Syst. Des. 41 (3) (2012) 269–294, http://dx.doi.org/10.1007/s10703-012-0142-8.
[49] D.A. Basin, V. Jugé, F. Klaedtke, E. Zalinescu, Enforceable security policies revisited, ACM Trans. Inf. Syst. Secur. 16 (1) (2013) 3, http://dx.doi.org/10.1145/2487222.2487225.
[50] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Computational analysis of run-time monitoring – fundamentals of Java-MaC, Electron. Notes Theor. Comput. Sci. 70 (4) (2002) 80–94, http://dx.doi.org/10.1016/S1571-0661(04)80578-4.
[51] A. Pnueli, A. Zaks, PSL model checking and run-time verification via testers, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), Proceedings of the 14th International Symposium on Formal Methods, FM 2006m Hamilton, Canada, August 21–27, 2006, in: Lect. Notes Comput. Sci., vol. 4085, Springer, 2006, pp. 573–586.
[52] A.P. Sistla, M. Zefran, Y. Feng, Runtime monitoring of stochastic cyber-physical systems with hybrid state, in: Khurshid and Sen [57], pp. 276–293, http://dx.doi.org/10.1007/978-3-642-29860-8_21.
[53] G. Rosu, On safety properties and their monitoring, Sci. Ann. Comput. Sci. 22 (2) (2012) 327–365.
[54] M. Viswanathan, M. Kim, Foundations for the run-time monitoring of reactive systems – fundamentals of the MaC language, in: Z. Liu, K. Araki (Eds.), Proceedings of the First International Colloquium on Theoretical Aspects of Computing, ICTAC 2004, in: Lect. Notes Comput. Sci., vol. 3407, Springer, 2004, pp. 543–556.

[55] B. Bonakdarpour, S.A. Smolka (Eds.), Proceedings of the 5th International Conference on Runtime Verification, RV 2014, Toronto, ON, Canada, September 22–25, 2014, Lect. Notes Comput. Sci., vol. 8734, Springer, 2014.
[56] E. Bartocci, R. Majumdar (Eds.), Proceedings of 6th International Conference on Runtime Verification, RV 2015 Vienna, Austria, September 22–25, 2015, Lect. Notes Comput. Sci., vol. 9333, Springer, 2015.
[57] S. Khurshid, K. Sen (Eds.), Proceedings of Second International Conference on Runtime Verification, RV 2011, San Francisco, CA, USA, September 27–30, 2011, Lect. Notes Comput. Sci., vol. 7186, Springer, 2012, Revised selected papers.